МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
Навчально-науковий інститут інноваційних освітніх технологій
Кафедра комп'ютерних інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
Савченко А.С.
«___»_____2020 р.

ДИПЛОМНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИЦІ ОСВІТНЬОГО СТУПЕНЯ МАГІСТРА

ЗА СПЕЦІАЛІЗАЦІЄЮ «ІНФОРМАЦІЙНІ УПРАВЛЯЮЧІ СИСТЕМИ ТА ТЕХНОЛОГІЇ»

Тема: «Реінженірінг програмних продуктів в гнучких технологіях з використанням патернів»

Виконавиця: студентка групи УС-201Мз Костанян Анастасія Олексіївна

Керівник: к.т.н., доцент Харченко Олександр Григорович

Нормоконтролер: Райчев І.Е.

Київ 2020

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Навчально-науковий інститут інноваційних освітніх технологій
Кафедра комп'ютерних інформаційних технологій
Напрям (спеціальність, спеціалізація): 12 «Інформаційні технології», 122 «Комп'ютерні науки», «Інформаційні управляючі системи та технології»
(шифр, найменування)

ЗАТВЕРДЖУЮ
Завідувач кафедри
Савченко А.С.
"_____" _____ 2020 р.

ЗАВДАННЯ

на виконання дипломної роботи студентки

Костанян Анастасії Олексіївни
(прізвище, ім'я, по батькові)

1. **Тема роботи**: «Реінженірінг програмних продуктів в гнучких технологіях з використанням патернів» затверджена наказом ректора від «06» жовтня 2020 р. № 1939/ст.

2. **Термін виконання роботи**: з 05.10.2020 р. по 31.12.2020 р.

3. **Вихідні дані до роботи**: наукові матеріали на тему гнучких технологій та рефакторингу, реінженірінгу та перепрограмування; уніфікована мова моделювання UML; CASE-засіб Rational Rose; методичні матеріали дисципліни «Стандартизація та сертифікація інформаційних управляючих систем».

4. **Зміст пояснювальної записки** (перелік питань, що підлягають розробці): вступ; гнучкі технології створення програмних продуктів, визначення та якість; архітектура програмної системи в гнучких технологіях; рефакторинг, реінженіринг та переписування програм в гнучких технологіях розробки; проектування програмного комплексу аналізу якості програмної архітектури та проведення рефакторингу; висновок.

5. **Перелік обов'язкового графічного матеріалу**: ієрархічне представлення задачі оцінювання архітектури, відношення патернів до компонентів архітектури, оцінка

архітектури на кожній ітерації, загальна тактика рефакторинга, спільний підхід до гнучкої архітектурної розробки з використанням виникаючої архітектури.

6. **Календарний план**:

№ з/п	Завдання	Термін виконання	Підпис керівника
1.	Огляд літератури і джерел за темою дипломного проекту.	5.10.2020 р. – 15.10.2020 р.	
2.	Створення плану та розробка 1 розділу – гнучкі технології створення програмних продуктів. Визначення та якість	16.10.2020 р. – 25.10.2020 р.	
3.	Розробка розділу 2: архітектура програмної системи в гнучких технологіях	26.10.2020 р. – 03.11.2020 р.	
4.	Розробка розділу 3: рефакторинг, реінженіринг та переписування програм в гнучких технологіях розробки.	04.11.2020 р. – 15.11.2020 р.	
5.	Створення та оформлення діаграм UML.	16.11.2020 р. – 22.11.2020 р.	
6.	Ознайомлення та аналіз методу аналізу ієрархій.	23.11.2020 р. – 02.12.2020 р.	
7.	Оформлення розділу 4: проектування програмного комплексу аналізу якості програмної архітектури та проведення рефакторингу.	03.12.2020 р. – 07.12.2020 р.	
8.	Створення доповіді та презентації.	08.12.2020 р. – 13.12.2020 р.	
9.	Оформлення та друк пояснювальної записки дипломного проекту.	14.12.2020 р. – 20.12.2020 р.	

7. Дата видачі завдання: 05.10.2020 р.

Керівник дипломної роботи _____ Харченко О.Г.
(підпис керівника) (П.І.Б.)

Завдання прийняв до виконання _____ Костанян А.О.
(підпис випускника) (П.І.Б.)

РЕФЕРАТ

Загальний обсяг пояснювальної записки до дипломного проекту роботи «Реінженірінг програмних продуктів в гнучких технологіях з використанням патернів» становить 112 сторінок, містить 26 рисунків, 21 літературних джерел, 13 таблиць та 6 формул.

Об'єкт дослідження: процеси модифікації архітектури програм при зміні вимог в гнучких технологіях програмування.

Предмет дослідження: метод реінженірингу програмної архітектури в гнучких технологіях звикористанням патернів.

Мета проекту: розробити метод забезпечення якості програмних продуктів у гнучких технологіях шляхом реінженірингу.

Методи дослідження: узагальнення поняття «виникаюча архітектура», аналіз архітектурного проектування у гнучких технологіях, аналіз методу аналізу ієрархій.

Отримані результати: розроблені діаграми програмного комплексу проведення реінженірингу архітектури у гнучких технологіях.

Результати дипломної роботи можуть бути використані при проведенні реінженірингу архітектури у гнучких технологіях на підприємстві.

Ключові слова: ГНУЧКІ ТЕХНОЛОГІЇ, АРХІТЕКТУРНЕ ПРОЕКТУВАННЯ, РЕФАКТОРИНГ, РЕІНЖЕНІРИНГ, МЕТОД АНАЛІЗУ ІЄРАРХІЙ, ПАТЕРНИ, ОЦІНЮВАННЯ ЯКОСТІ, ЗМІНЮВАНІСТЬ, АЛЬТЕРНАТИВНІ АРХІТЕКТУРИ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

ASD – Agile Software Architecture

SA – Software Architecture

SQA – Software quality assurance

XP – eXtreme Programming

MVC – Model View Controller

MVP – Model View Presenter

DAO – Data Access Object

ООП – Об'єктно-орієнтоване програмування

ПС – Програмна система

ПЗ – Програмне забезпечення

MAI – Метод аналізу ієрархій

ЖЦ – Життєвий цикл

ВСТУП

Процес розробки програмного забезпечення (ПЗ) суттєво змінюють та покращують такі методи гнучких технологій як екстремальне програмування (XP), гнучкий підхід управління проектами (Scrum), Kanban та інші. Прихильники даних методів запевняють, що якість гнучких програмних проектів є закономірним результатом застосованого методу через саму природу таких методів. В результаті очікується, що гнучка система забезпечення якості розробки програмного забезпечення (ASDQA) буде вбудована в гнучкі програмні процеси, поки методи забезпечення якості програмного забезпечення (SQA) вбудовані протягом всього життєвого циклу(ЖЦ), від початку встановлення вимог і до останнього випуску включно. Отже, гнучкі методи дозволяють інакше розглянути контролювання якості при розробці програмного забезпечення.

Гнучкі технології мають дозволити випускати ПЗ у короткі терміни, з меншою кількістю помилок, з певними бюджетними обмеженнями, використовувати непостійні та мінливі вимоги протягом усього ЖЦ розробки програмного продукту. Такий поступовий спосіб розробки дозволяє забезпечити механізм зворотного зв’язку, завжди розуміти вимоги замовника і залучати його у процес прийняття рішень, що в результаті призводить до задоволення кінцевим продуктом. Таким чином, гнучкі методи також змушують замовників допомагати покращувати та забезпечувати якість продукту. Використання таких практик як парне програмування, просте планування та проектування, метафора, невеликі випуски, спільне володіння кодом, короткі ітераційні цикли та постійні ітерації потенційно посилює забезпечення якості.

Тому метою першого розділу є розширити розуміння та дати всебічне визначення гнучким технологіям та пояснити, вирішити проблеми, пов’язані з адаптацією гнучких технологій, а також яким чином гнучкі методи можуть удосконалити забезпечення якості ПЗ, порівняно з традиційними підходами. Дане розуміння також представить можливість виявити, що означає якість у гнучких технологіях.

Коли частини архітектури програмної системі можуть змінюватися та коли розробляються подібні системи з різними сценаріями використання, важливу роль займає змінюваність – очікувані і попередньо заплановані зміни. Змінюваність включена та відображається в архітектурі ПЗ у великій кількості сучасних систем, наприклад, само-адаптивні системи, системи, які підтримують динамічні веб-сервіси тощо. Тобто під час проектування архітектури ПЗ необхідно пам’ятати про керування змінюваністю – процеси, які стосуються ідентифікації, обмеження, впровадження та керування змінюваністю.

Важливою є гнучка парадигма – гнучкі методи, які створені на основі принципів маніфесту, та практик, які забезпечують адаптацію до змін. Змінюваність зручно використовувати у випадках проектування систем, коли потреби клієнтів або контексти недостатньо відомі, а гнучку розробку – з будови програмних рішень ще до того, як продукт буде цілковито зрозумілим.

Тож за допомогою поєднання при проектуванні архітектури гнучкої парадигми та змінюваності можливо досягти гнучкої адаптивної програмної архітектури, і відповідно системи. Такі зміни забезпечують доступність інформації про змінність на різних етапах розробки – при тестування та реалізації системи. Тому такий підхід поєднання може бути корисний проектам з часто змінюваними вимогами, важливістю постійного зворотного зв’язку з клієнтами, короткими термінами та графіками, необхідністю зосередитися у більшій мірі на розробці ПЗ, а не на документації та плануванні, що дозволить керувати змінюваністю за короткий проміжок часу і з невеликої кількістю зусиль.

Останнім часом популярною темою в гнучкій методології є тема виникаючої архітектури, яка тісно пов'язана з суперечками, чи архітектор є важливою і необхідною частиною в гнучких проектах або всі архітектурні роботи можуть бути виконані командою розробників. Тому інша частина цього розділу розглядатиме виникаючу архітектуру, як і в якій мірі претензії на виникаючу архітектуру є істинними. Тому спочатку буде розглянута мета, діяльність і завдання архітектурної роботи. Згодом буде проаналізовано, чи підходить виникаюча архітектура для заміни різних видів діяльності та цілей архітектурної роботи, зважаючи на її мету.

Під час обговорення, сильні та слабкі сторони явних і виникаючих архітектурних робіт будуть протиставлятися одна одній, і буде запропонований спільний підхід.

З кожним роком у сфері програмування відбувається все більше змін, які в результаті спричиняють до труднощів, які зустрічаються в архітектурі та які усунути стає складніше та дорожче по витратам. Прикладами таких змін можуть бути: зміна технологій розробки та інфраструктури, створення/введення/перегляд додаткових вимог, які і призводять до виникнення помилок(багів) і прийнятих помилкових рішень.

Але зосереджуватися важливо не на змінах, які відбуваються в системі, а саме на архітектурі ПЗ, так як якщо вона буде спрямована на внесення нових функцій, то вони можуть настільки збільшити і розширити даний проект, що в кінці кінців він стане некерованим і супроводжувати такий програмний продукт стане неможливим. Це стосується і певних випадків розробки архітектури, коли кожне помилково прийняте рішення призводить до незворотного результату та наслідків.

Аби уникнути такої долі проекту, обов'язковим фактором в архітектурі ПЗ є періодичне чергування проектних робіт з ітеративною оцінкою архітектури та проведення рефакторингу – метод покращення внутрішньої структури ПЗ без зміни зовнішньої поведінки системи. Якщо впровадити систематичне проведення рефакторингу у свій проект, це дозволяє розробникам ПЗ використовувати рішення, які є вже перевіреними та які приносять успішний результат. Тобто проведення оцінювання архітектури та рефакторингу допомагає уникнути ерозії проекту.

Мета рефакторингу покращити область, яка в цілому може поліпшити архітектуру. Для того аби визначити проблеми цієї області, а отже, і архітектури, і в подальшому провести рефакторинг було введено поняття «код з душком».

Покращити якість внутрішньої архітектури можна за допомогою моделей та патернів рефакторингу. Одним із шляхів проведення рефакторингу архітектури – використання патернів, коли архітектура складається із стандартних компонентів і внесення змін відбувається пошарово. Однак існує багато альтернативних шаблонів, тому необхідно їх якось оцінювати і вибрати кращий. Є декілька методів

оцінювання, найбільш конструктвний і обгрунтваний, на наш погляд – метод аналізу ієрархій.

І в останньому розділі створено проектування програмного комплексу аналізу якості архітектури та проведення рефакторингу, описано процес створення альтернативних архітектур, оцінювання їх та прийняття рішень, що є досить актуальним, так як це допомагає визначити якість програмного продукту. Тому метою цієї роботи є проектування програмного комплексу аналізу якості архітектури та проведення рефакторингу, який вирішить задачі перепроектування архітектури при внесенні змін у вимоги і як результат забезпечить високу задоволеність замовника, який постійно контактує з розробниками і вносить свої корективи у розробку програмного продукту.

РОЗДІЛ 1

ГНУЧКІ ТЕХНОЛОГІЇ СТВОРЕННЯ ПРОГРАМНИХ ПРОДУКТІВ. ВИЗНАЧЕННЯ ТА ЯКІСТЬ

Гнучкі методи розробки ПЗ дозволяють не тільки задовольнити, перевірити та підтвердити вимоги замовника, а й відкрити нові горизонти, розширити концепції забезпечення якості ПЗ. Гнучка розробка ПЗ забезпечує зміну вимог аж до рівня випуску програмного продукту. Наразі існує література, яка в достатній мірі описує дану технологію від різних авторів методик та декількох практикуючих спеціалістів. Але не вистачає інформації, яка б описувала реальні приклади застосування даної методології у реальних проектах за останні роки, щоб допомогло зрозуміти та вдосконалити дану методологію в майбутньому. Деякі спеціалісти через нерозуміння основних концепцій, що закладені у гнучкій методології, не приєднуються до такого способу розробки ПЗ. Тому метою даного розділу є розширити розуміння та дати всебічне визначення гнучким технологіям та пояснити, вирішити проблеми, пов'язані з адаптацією гнучких технологій, а також яким чином гнучкі методи можуть удосконалити забезпечення якості ПЗ, порівняно з традиційними підходами. Дане розуміння також представить можливість виявити, що означає якість у гнучких технологіях.

1.1 Визначення гнучких технологій

У лютому 2001 року 17 незалежних фахівців, які практикували декілька методик розробки ПЗ, створили асоціацію Agile Alliance, що була направлена на формалізацію гнучких технологій. Також у лютому того ж самого року був створений основний документ маніфест гнучкої розробки ПЗ Agile Manifesto, що включав у себе основні принципи даної методології [2].

У червні 2002 року після першої масштабної зустрічі з гнучких технологій узагальнили остаточне визначення гнучких методологій – група процесів розробки ПЗ, які є самоорганізуючими, ітераційними, поступовими та виникаючими [2]:

1) *самоорганізація*: даний термін у рамках гнучкої методології означає, що команда має можливість самостійно організовувати себе для кращого майбутнього результату, на відміну від звичайного підходу, коли команда має організувати себе в залежності від навичок та відповідних завдань і необхідності звітуватися керівництву в ієрархічній структурі. Концепція «самоорганізації» відносно іноземна в управлінні науковими процесами. Команда має вирішити, як найкраще організувати такі процеси як робочий час, взаємодія в команді, звіти та зустрічі про хід виконання та роботи, динаміка команди тощо. Відповідно такий підхід вимагає від керівників проектів змінювати парадигму управління і від членів команди поводитися професійно та поважати один одного у ситуаціях, коли певні забов'язання були невиконані. У такому випадку керівнику проекту необхідно посприяти вирішенню проблем, які виникають у команди, де це є можливим. Тобто підхід самоорганізації очікує від команди розробників та керівництва проекту чіткої комунікативної політики;

2) *ітеративний*: даний термін походить від слова «ітерація», що має сумарне та тотальне значення слова «повторення». Якщо розглянути даний термін у рамках гнучких технологій, то це не просто «повторення», це певний підхід до вирішення програмної проблеми методом пошуку певних послідовних наближень до рішення, які повторюються, починаючи від перших мінімальних вимог, закінчуючи зміненням функціональності всіх підсистем з кожним новим релізом, оскільки кожного такого випуску оновлюються вимоги. Тобто даний такий підхід дозволяє поступово вирішити певну глобальну проблему за декілька кроків, а не за один кадр, як це прийнято у традиційних методах. Тому поняття ітерацій тісно пов'язано з поняттям поступового розвитку. Так як процес розробки ПЗ є досить складним та в ньому досить швидко змінюються вимоги, то такий ітеративний підхід є дуже актуальним для вирішення проблем, які зустрічаються на шляху будови програмного продукту;

3) *поступовий*: даний підхід полягає у тому, що система розділена на підсистеми за функціональністю і розроблена таким чином, що надає можливість збирати вимоги і використовувати їх у розробці інших підсистем. Також підхід також передбачає додавання нової функціональності з кожним новим випуском, і корисні функціональні можливості додаються до того моменту, коли повна система не буде реалізована;

4) *виникнення*: даний термін випливає з термінів, вказаних вище. Система може виходити з ряду приростів через поступовий підхід до розвитку. На основі концепції самоорганізації формується метод роботи у процесі роботи команди розробки. У результаті, по мірі появи системі і методу роботи виникне новий характер гнучких методологій. Тобто гнучка розробка ПЗ фактично є досвідом та результатом навчання та роботи кожного окремого проекту, так як кожен проект є особливим та індивідуальним, застосовуючи самоорнанізуючі, поступові, новітні та ітеративні методи.

Визначення гнучких методологій можна зобразити схематично (рис. 1.1):

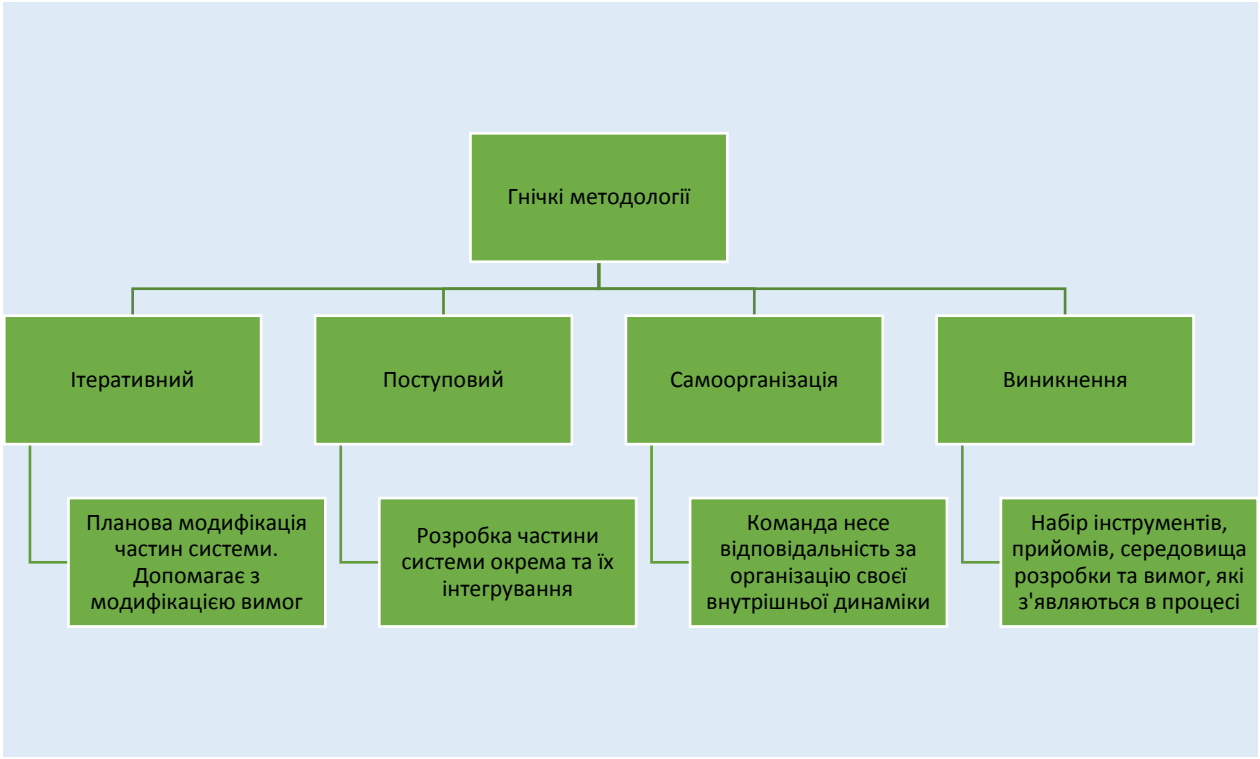

Рис. 1.1. Визначення гнучких методологій

Також гнучкі методології можна визначити відповідно до того, як практикуючі спеціалісти зрозуміли дану методологію на практиці. Таким чином, термін «agile» включає в себе поняття спритності, активності, гнучкості, готовності до руху, регульованості та моторності в русі [2]. Кожне з даних слів пояснимо в умовах гнучкого розвитку:

1) *спритність* – це слово означає, що доставка продукту у контексті гнучкої розробки ПЗ має бути швидкою. Випуск релізів підсистем має відбуватися протягом певного періоду, зазвичай, це від 1 до 4 тижнів;

2) *активність* – основний сенс даного слова полягає у тому, щоб замість того аби постійно планувати, що іноді навіть займає більшу частину часу на розробку ПЗ, необхідно активне написання коду;

3) *гнучкість* – дане слово пояснює, що правила та процеси у гнучкому способі розробки ПЗ можна легко адаптувати та змінювати залежно від заданих ситуацій, не обов'язково порушуючи їх;

4) *готовність до руху* – у контексті гнучкої методології цей термін означає, що необхідно зменшити всі механізми та види діальності, які якимсь чином можуть уповільнити швидкість розвитку;

5) *регульованість* – вимоги, код, дизайн та архітектура повинні мати можливість видозміни для виконання потреб замовника. Тобто у технології та наборі діяльності має бути дозволено вносити зміни, що забезпечить сприятливий процес розробки;

6) *рухливість* – головним поясненням даному терміну є наявність розумових навичок в діяльності розробки коду, які допоможуть розробникам у вирішенні програмних питань та динаміки команди.

Визначення гнучких методологій згідно визначення практикуючих спеціалістів можна зобразити схематично (рис. 1.2):

Гнучкі методології	Спритність	Випуск продукту має бути швидким
	Активність	Необхідність активного написання коду замість планування
	Гнучкість	Легка адаптація до змін без порушення правил
	Готовність до руху	забезпечити максимальну швидкість розвитку
	Регульованість	Можливість видозміни для виконання потреб замовника
	Рухливість	Розумові навички в діяльності розробки коду

Рис. 1.2. Визначення гнучких методологій згідно практикуючих спеціалістів

Гнучкі методології також трактував Бек [4]. За його словами, такі методології – це гнучкий, науковий, легкий та ефективний, передбачуваний, з низьким рівнем ризику та цікавий спосіб розробки ПЗ. Перелічені слова пояснимо в рамках концепції гнучкого розвитку:

1) ефективні передбачає виконання лише тієї роботи, яка випустить потрібний продукт з меншими витратами;

2) легкі означає мінімізувати все, що можливо, у процесі розробки(це може бути створення вимог або документації), аби збільшити ефективність та швидкість самої розробки. Принцип мінімізування певних процесів є досить суперечливим, так як є припущення, що гнучка методологія не має ніякої, наприклад, документації, що дійсно рідко, але зустрічається у практиці;

3) з низьким рівнем ризику означає, що часто витрачається забагато зусиль на абстрагування проблемного простору задля управління ризиком і цього необхідно уникати. Треба розуміти, що методології розробки ПЗ створені для зменшення ризиків відмови проекту, тому такий підхід не означає, що управління ризиками не відбувається, але в той самий час дозволяє не концентрувати всі сили на них;

4) передбачувані – не означає, що такі частини як дизайн, архітектура та дизайн ПЗ є передбачуваними. Цей термін базується на тому, що гнучкість дозволяє розробляти ПЗ та користуватися певними інструментами такими звичними способами, які досвідчені розробники можуть визначити на основі свого досвіду та знать, тобто передбачити;

5) наукові – означає, що принципи гнучкої розробки ПЗ засновані на наукових засадах, які є перевіреними та обгрунтованими. Але це не означає, що такі принципи не розвиваються, навпаки, наука поповнює докази про гнучкі процеси, так як велика кількість практикуючих спеціалістів не займаються написанням наукових праць і не проводить такого роду дослідження;

6) цікавий спосіб розробки ПЗ – даний підхід забезпечує розробникам бути більш творчими та ініціативними, витрачати більшу частину часу на написання коду, який їм подобається, відповідно на написання хорошого коду, який працює.

Визначення гнучких методологій згідно визначення Бека можна зобразити схематично (рис. 1.3):

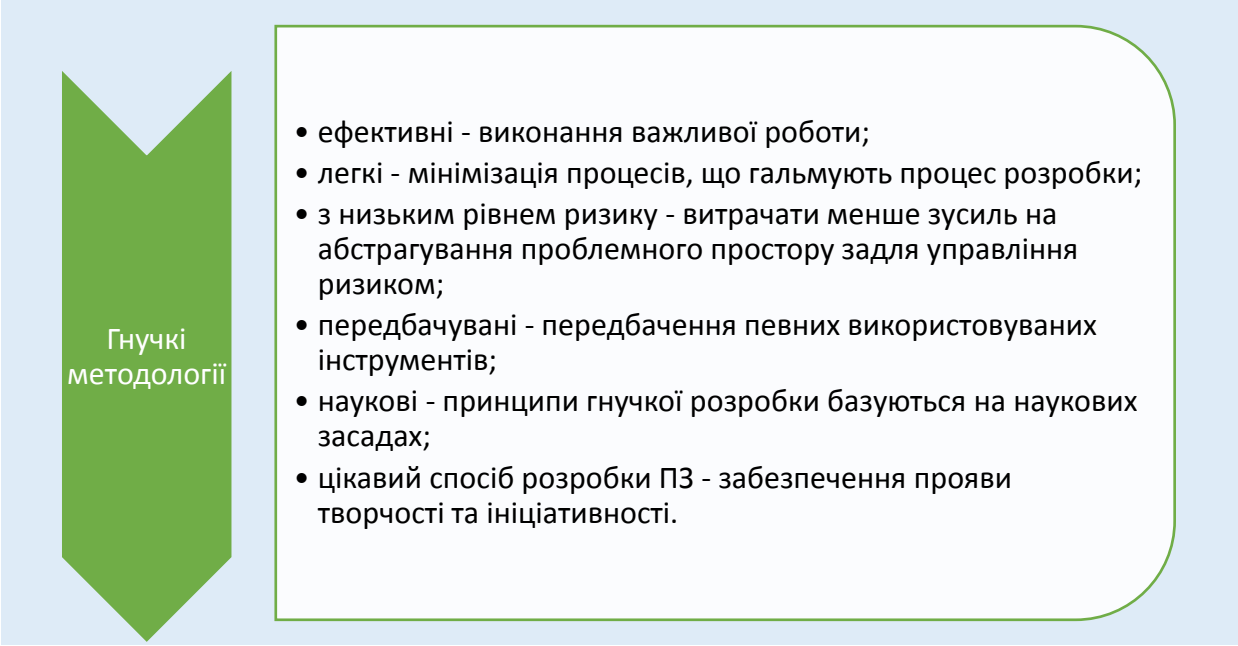

Рис. 1.3. Визначення гнучких методологій згідно Бека

В загальному, гнучкі технології почали набирати своєї популярності тоді, коли фахівці-практики зрозуміли, що витрачають забагато часу на документообіг та

розроблення підходів до розробки і почали орієнтуватися на підходи, які сконцентровані на клієнтах, їх потребах та на самій розробці з меншим об’ємом документообігу [6], [12], [9]. Але іноді хибно уявляють процес планування у гнучкій методології, що його фактично немає або його замало, але на практиці це не так. Планування може бути навіть точнішим, аніж у традиційних процесах, яке проходить обов’язково для кожного приросту і з концепцією зворотного зв’язку, що забезпечує менші ризики: глибоко недосконалої архітектури, непорозуміння з функціональними вимогами, коли команда не розуміє обрану технологію, небажаного інтерфейсу для користувача, невірних моделей проектування та аналізу тощо. Крім таких ризиків, важливим принципом гнучкої методології є контроль таких передумов як архітектура, вимоги, плани та проекти, повторюваності процесів, тобто якщо вони можуть бути передбачуваними, то їх можливо швидше і ефективніше стабілізувати. Повторюваність процесів, наприклад, є важливою частиною традиційних методологій, що значно їх обмежує у розвитку [9], [10].

Тобто гнучкі технології сильно відрізняються від стандартних поглядів на розробку програмного забезпечення своїми зручними принципами та концепціями, які вказані у маніфесті, і цим привертають аудиторію розробників, які готові спробувати принципово інший підхід, але не менш ефективний.

Згідно визначень, які подані вище, можна в повному вигляді зрозуміти що таке гнучка методологія і отримати розуміння, що таке рухливість з погляду певних конкретних концепцій програмної інженерії: моделювання програмних процесів, забезпечення якості ПЗ, управління програмними проектами та вдосконалення програмних процесів.

1.2 Оцінювання якості гнучких процесів

Наведемо декілька цитат різних авторів, які визначають термін якості по різному, але в результаті можна зробити висновок, що всі вони мають головні спільні аспекти, які формулюють одне загальне визначення.

За словами Джурана, якість – це придатність для використання, що означає наступні дві речі: «(1) якість складається з тих характеристик товару, які відповідають потребам споживачів і тим самим забезпечують задоволення товару. (2) Якість складається із позбавлення від недоліків» [12].

Філіп Кросбі, який розробляв та викладав концепції управління якістю, вплив яких можна знайти у стандарті ISO 9000: визначає якість як відповідність вимогам та нульові дефекти.

Деякі інженери-програмісти визначили якість ПЗ таким чином:

1) Meyer визначає якість ПЗ відповідно до адаптованої кількості параметрів якості – це правильність, надійність, розширюваність, багаторазове використання, сумісність, ефективність, портативність, цілісність, перевіреність та простота використання [14];

2) Прессмен, який вивів своє визначення з Кросбі, визначає якість як "відповідність чітко заявленим функціональним вимогам, чітко задокументованим стандартам розробки та неявним характеристикам, які очікуються від усього професійно розробленого ПЗ" [16];

3) Соммервілл (2004) визначає якість ПЗ як процес управління, що займається забезпеченням того, щоб ПЗ мало низьких дефектів і що воно відповідає необхідним стандартам ремонтопридатності, надійності, портативності тощо [18].

Тобто головною думкою у всіх цих визначеннях є пояснення якості як відповідності чітким параметрам та фунціональним вимогам, які забезпечують максимальне позбавлення від недоліків та дефектів та максимальне задоволення від продукту споживачами.

Після означення терміну якості виникає питання, чи можливо оцінити забезпечення якості у гнучких процесах. Це можливо за рахунок:

- якщо є розуміння і знання якості гнучких процесів;
- визначення методів якості для певних гнучких методологій;
- визначення нових шляхів покращення якості гнучких процесів.

Була розроблена техніка порівняння, яка мала забезпечити аналіз традиційної моделі розвитку водоспадів та якість гнучких процесів:

– було визначено, що головні фактори управління якістю – це перевірка та забезпечення якості;

– було визначено методи, які дозволяють покращити управління якості в гнучких процесах;

– швидкість забезпечення якості гнучких процесів не відрізняється від швидкості забезпечення якості традиційних.

У результаті було виявлено, що якість у гнучкій методології існує і її можливо досягти трохи інакше, аніж у традиційному підході. Але незважаючи на те, що головним посилом автора було визначити чи є гарантія якості, йому не вдалося зрозуміти, де вона є кращою – у традиційних чи гнучких процесах. Але завжди знайдуться прибічники тієї чи іншої думки, які завжди будуть захищати свою методологію.

Важливість створеної техніки порівняння важко недооцінити, так як завдяки ній можливо досягти тієї високої якості гнучких технологій і більш того, в подальшому удосконалити її та виявити ще можливі більш підходящі практики.

Таблиця 1.1

Техніка порівняння параметрів якості ПЗ для гнучких методів

Параметри якості ПЗ	Гнучкі методи	Можливі вдосконалення
Правильність	Необхідність написання коду з мінімальними вимогами. Уточнення та вимоги можна отримати у ході безпосереднього спілкування із замовником. Замовнику дозволено змінювати вимоги. Проведення тестів.	Можливо використовувати формальні та загальні вимоги.
Надійність	Не вирішується безпосередньо у гнучій розробці.	Включайте можливі екстремальні умови в вимоги.

Продовження таблиці 1.1

Розширюваність	На що важливо звернути увагу у першу чергу – це прагнення до ідеальної архітектури, гарний дизайн та технічна досконаліть.	Використання методів архітектурного моделювання ПЗ.
Багаторазе використання	Не використовувати повторно гнучкі продукти [16], [17].	Розробка шаблонів(патернів) для гнучких продуктів.
Сумісність	Загальна особливість всіх ОО продуктів.	Не додавати функції, які можуть не знадобитися, але реалізуються заради сумісності, це суперечить принципу простоти.
Продуктивність (ефективність)	Використовуйте вправні стандарти кодування.	Заохочуйте проекти, засновані на найбільш ефективних алгоритмах.
Переносність	Практика безперервної інтеграції в ХР.	Деякі гнучкі методи не вирішують проблеми розгортання продукту.
Своєчасність	Найсильніша точка гнучкості, Короткі цикли, швидкі випуски(релізи) тощо.	–
Цілісність	Не вирішується безпосередньо у гнучій розробці.	–
Перевіреність	Тестування – ще одна перевага гнучкості.	–
Простота використання	Так як у гнучкій розробці присутній зворотній зв’язок від замовника, то користувач зазвичай бажає просту у використанні систему.	Найпростіший дизайн для найменш кваліфікованого користувача.

Параметри техніки порівняння можуть здатися досить абстрактними. Але в даній техніці дані гнучкі методи реалізують ці параметри так чи інакше. Вдосконалення, які наявні у третьому стовпці у табл. 1.1, були виявлені при аналізі роботи гнучких практик, які могли бути навіть не задекоментовані, але використовувалися при аналізі. Це наводить на думку, що весь цей процес досить об’єктивний, але в цьому і є головна цінність такої інформації. Фахівці-практики

використовують велику кількість знань і на основі свого досвіду формують техніки, які і не дотримуються жодної формальної методології збору даних, але головне успішно застосовуються. Але для сбалансування фактів фахівцям все ж таки необхідно застосовувати і інші методи дослідження, щоб не з'являлося упереджень з приводу достовірності поданої інформації.

Для того щоб забезпечити якість ПЗ згідно техніки наявної у табл. 1.1, необхідно щоб у ЖЦ розробки ПЗ був реалізований кожен із параметрів, що є у таблиці. Коротко дамо визначення кожному параметру [14]:

– *правильність*: визначені вимоги визначають як саме має працювати система;

– *надійність*: при певних екстремальних умовах в системі має бути наявна продуктивність системи, що відповідно допомнює правильність;

– *розширюваність*: система, що легко змінюється відповідно до нових вимог;

– *багаторазе використання*: ПЗ, яке містить підсистеми, які можливо застосувати для реалізації інших продуктів;

– *сумісність*: ПЗ, яке містить підсистеми, які без перешкод пов'язуються з іншими підсистемами;

– *продуктивність(ефективність)*: менші вимоги до апаратних ресурсів(пропускна здатність, час процесора, пам'ять);

– *переносимість*: легке завантаження ПЗ на різних апаратних та програмних платформах;

– *своєчасність*: випуск(реліз) ПЗ перед тим або саме тоді (головне не після того), коли цього потребують користувачі;

– *цілісність*: важлива складова, яка описує необхідність можливості ПЗ захищати дані від несанкціонованого доступу;

– *перевіреність*: можливість тестування;

– *простота використання*: можливість невимушено користуватися системою користувачам незважаючи на їх кваліфікацію та вміння.

1.3 Методика оцінювання ефективності гнучких технологій

Ширший погляд на гнучкі технології відкрив розуміння того, що гнучкі методи – це сукупність певних процесів, які значно зменшують час на розробку ПЗ та допомагають миттєво підлаштовуватися під зміну вимог. Необхідний час для того, щоб ці відносно нові методи переросли у справжні установлені стандарти гнучкого моделювання програмних процесів.

Майбутнє гнучких технологій полягає в тому, що ідеї, які покладені у цю технологію дуже актуальні і полягають у тому, щоб мінімізувати документацію, а простота бажаної системи була перевагою. Також велика цінність надається розробникам, тобто керівники проектів мають намагатися надати сприятливе середовище розвитку своїм підлеглим. Замість навантаження робітників обчисленнями та будуванням графіків для ілюстрації плану, графіка робіт за будь-яким проектом, керівник має спростити процес отримання інформації про хід та проблеми проекту, наприклад, спілкуванням один з одним, а також надавати простір для знаходження кращого методу виконання своїх робіт – тобто довіряти людям та їх рішенням.

Особливістю гнучкої методології є те, що між її методами немає конкретних меж, так як вони дуже подібні між своїми процесами і мають однакові принципи у своїй суті, і можна застосовувати практики інших, якщо це доречно у данній ситуації [12]. Але наявна тут методика оцінювання гнучких технологій показує, що LD розглядає розробку ПЗ за допомогою метафори виготовлення та виробу, Scrum – з використанням метафори керуючої інженерії, XP – як соціальну діяльність, коли практики працюють поряд, а ASD – як теорію складних самоадаптаційних систем [4].

Аналіз декілька обраних існуючих гнучких технологій було обрано та узагальнено у табл. 1.2-1.5. У табл. 1.2-1.4 обрані лише певні елементи, які є досить суб’єктивними, але які були обрані для аналізу та виявлення подібності різних гнучких технологій. Такий аналіз та розуміння подібності гнучких методів допомагає фахівцям-практикам, які тільки почали знайомитися з гнучкою

методологією, яку саме методологію їм обрати. Але це не означає, шо методологія, яка була обрана у певному проекті, найбільш підходяща та призведе до успіху ваш проект, так як велика частина організацій просто не може використовувати для різних проектів різні методології. Тому важливо розуміти, що необхідно вірно обрати технологію, яка підходить саме вашому проекту, яка зможе призвести до якісного результату виробництва продукту. Але в той же час виникає питання: тобто для кожної методології має бути певний набір фахівців, які можуть забезпечити якісне її використання? Це досить непрактично. Але і використовувати одну методологію і очікувати успіху від всіх проектів – це програшний варіант. Тому наявна в даному розділі методика оцінювання дозволяє адаптувати процес розробки ПЗ відповідно до загальних практик різних гнучких методологій, що дозволяє мінімізувати витрати компанії без придбання окремих технологій.

Для того, щоб визначити загальні практики і зрозуміти ціль метології, необхідно проаналізувати кожну гнучку методологію, в результаті чого буде можливо виявити основні проблеми і ризики. Тобто дана методика оцінювання також дозволяє виявити основні проблемні місця, які може вирішити та чи інша гнучка технологія: XP в загальному вирішує питання як і коли перевірити код; Scrum вирішує проблеми управлення проектами – ефективне спілкування в рамках проекту; наприклад, Crystal займається більш загальними проблемами, такі як ефективність розміру команди або методології, критичність проекту. Тобто кожна методологія має приціл на певну проблему і відповідно вирішує її відповідно до своїх концепцій.

Крім того, оцінка кожної методології дозволяє визначити основні практики і види діяльності, хоча деякі з них відповідають однаковим принципам. Ця інформація однозначно допоможе користувачам технології визначити практики, які вони будуть використовуватися в тій чи іншій ситуації. Тому в цілому неважливо, яка практика була обрана, користувач використовує певні принципи тієї методології, яка може допомогти йому в даному випадку.

Також за допомогою цієї методології оцінки можна зрозуміти, який же результат очікується від проекту, так як користувач має певне очікування від

використання конкретної методології і якщо результат є не достатнім або не зрозумілим, то користувач має право відмовитися від певної технології або взяти на себе відповідальність і проблеми, які можуть виникнути в результаті будуть повністю на плечах того, хто її обрав. Якщо методологія забезпечує доставку коду, а справжня суть засновується на створенні набору артефактів дизайну, що доставляються через гнучке моделювання – це звісно може призвести до не очікуваного результату.

В решті решт, методика розкриває важливі практичні знання її авторів. У табл. 1.2-1.4 наявний аналіз конкретних гнучких методологій, який допоможе використовувати ці знання для інших методологій.

1.3.1 Аналіз «Scrum»

Дана гнучка методика Scrum використовувалася довше, аніж інші методики і є найбільш використовуваною порівняно з іншими. Основною метою цієї методики є те, що «певні повторювані процеси працюють лише для вирішення певних повторюваних проблем із певними повторюваними людьми у певних повторюваних середовищах», що є неможливим. Для того, аби вирішити таку проблему «визначених повторювальних процесів», Scrum ділить проект на ітерації, які називаються спринтами. Спочатку прописується функціональність для такого спринта і вже потім команда має його реалізувати та випустити реліз продукту. Суть під час такої ітерації у стабілізації вимог. Головною концепцією Scrum є управління проектами. Запозичений термін Scrum у Регбі: "Scrum виникає, коли гравці кожної команди тісно стикаються ... намагаючись просунутись в ігрових умовах". Методика аналізу до Scrum відображена у табл. 1.2 [11].

Таблиця 1.2

Методика аналізу до Scrum

Елементи	Опис
Справжня метафора життєдіяльності	Керування технікою
Фокусування	Керування процесом розвитку.
Межа застосування	Команд менше 10, але кількість збаульшується для більших команд.
Процес	Етап 1: планування, відставання продукту та дизайн. Етап 2: відставання спринту, спринт. Етап 3: тестування системи, інтеграція, документація та випуск.
Результати	Робоча система.
Прийоми	Спринт, відставання в scrum (випадки використання письма).
Автор методики (два)	1. Розробник ПЗ, менеджер продуктів та галузевий консультант. 2. Розроблені мобільні додатки на відкритій платформі технологій. Розробник компонентів технології. Архітектор передових систем робочого процесу в Інтернеті.

1.3.2 Аналіз методології «Lean Development»

Методологія розглянута вище, була розроблена Боб Шаретт і який опирався на успіх цієї методології в 1980-х в автомобільній промисловості, але Scrum – це не певний процес, це набір практик управління проектами. Боб Шарет вважає, аби бути дійсно гнучким, треба змінити спосіб роботи компаній зверху вниз (Mnkandla et al., 2004b). Lean development (LD) – бережлива розробка ПЗ, що заснована на бережливому мисленні та бережливому виробництві, постійному прагненні до

усунення всіх видів втрат (интернет). Методика аналізу на LD відображена у таблиці 1.3 [15].

Таблиця 1.3

Методика аналізу до LD

Елементи	Опис
Справжня метафора життєдіяльності	Виробництво та розробка продукції.
Фокусування	Керування змінами. Керування проектами.
Межа застосування	Немає конкретного розміру команди.
Процес	Не має процесу.
Результати	Надає знання для керування проектами.
Прийоми та інструменти	Бережливі технології виготовлення.
Автор методики	Науковий співробітник Центру морських підводних систем США, автор книг та робіт з програмної інженерії, консультативна рада з управління проектами.

1.3.3 Аналіз екстремального програмування

Екстремальне програмування (XP) – це методологія для невеликих та середніх команд, що розробляють ПЗ, яка базується на стрімко мінливих вимогах. У іншому визначенні екстремального програмування Бек розширив поняття XP, щоб додати такі елементи, як розмір команди та обмеження ПЗ таким чином [4]:

– *XP легкий*: необхідно робити тільки те, що створить цінність для клієнта.

– *XP підлаштовується до стрімко мінливих вимог*: навіть для проектів зі стабільними вимогами можна використовувати XP.

– *XP розглядає обмеження в розробці ПЗ*: він безпосередньо не стосується управління фінансовими проблемами проекту, маркетингом, портфелем проектів, продажами або операціями.

– *XP працює з командами будь-якого розміру*: XP може масштабуватися для великих команд.

Таблиця 1.4

Методика аналізу до XP

Елементи	Опис
Справжня метафора життєдіяльності	Соціальна діяльність, де розробники працюють разом.
Фокусування	Технічні аспекти розробки ПЗ.
Межа застосування	Менше десяти розробників в кімнаті. Масштабується до більших команд.
Процес	Етап 1: Написання історій користувачів. Етап 2: Оцінка зусиль, визначення пріоритетності історії. Етап 3: Кодування, тестування, тестування інтеграції. Етап 4: Невеликий випуск. Етап 5: Оновлений випуск. Етап 6: Остаточний випуск (Abrahamsson et al, 2002).
Результати	Робоча система.
Прийоми та інструменти	Парне програмування, рефакторинг, тестова розробка, постійна інтеграція, метафора системи.
Автор методики (два)	1. Розробник ПЗ(Smalltalk) . Сильно віруючий у спілкуванні, роздумах та інноваціях. Шаблон для ПЗ. Перший тест розробки. 2. Розробник ПЗ(Smalltalk). Директор з досліджень та розробок. Розробив Wiki. Розроблена структура для

	інтегрованого тестування (Fit).

Перший етап розробки ПЗ за допомогою XP починається з написання історій замовником для опису функціональності ПЗ, які складаються з малих одиниць функцій, на реалізацію та тестування яких необхідно приблизно 1-2 тяжня. Потім розробники надають кошторис для сторіс, замовник переглядає та вирішує, які історії необхідно робити першими виходячи з вартості. Процес розробки виконується поступово та ітераційно. Команда розробників випускає реліз замовнику з працюючою історією кожні два тижні. Після цього замовник обирає наступну історію, яка буде виконана за наступні 2 тижні. Таким чином, ітераційно, по частинах, в системі збільшується функціонал. Методика аналізу на XP відображена у табл. 1.5.

Далі представимо у табл. 1.2-1.4 опис кожного методологічного елементу:

1) *справжня метафора життєдіяльності методології* – данний елемент стосується фундаментальної моделі(метафори) та обставин, які породили початкову ідею методології. Наприклад, процес збирання мурашника мурахами може надихнути використати цей самий принцип у розробці ПЗ;

2) *методологічний фокус* – цей елемент посилається на певні аспекти процесу розробки ПЗ. Наприклад, гнучке моделювання досліджує як моделювати складні та великі проекти гнучким методом та направлене на аспекти дизайну процесу розробки ПЗ;

3) *методика межі застосування* – даний елемент описує деталі межі розробки технології, тобто допомагає зрозуміти якими завданнями буде керувати методологія. Даний елемент при розробці ПЗ є необхідним при визначенні чисельності команди;

4) *методологічний процес* – цей елемент відображає, як методологія моделює реальність, що дає розуміння реального світосприйняття процесу розробки ПЗ. Модель може зафіксувати та дати зрозуміти суть проблемних областей, забезпечити засіб комунікації, бути відображена у ЖЦ чи процесі розробки ПЗ [3];

5) *результати методології* – даний елемент дає розуміння того, які результати можна очікувати від методології. Для кожної методології існує свій кінцевий результат і тому він у кожній методології буде відрізнятися один від одного, тому користувачу необхідно обирати саме ту методологію, результат якої задовольнить його цілі [20];

6) *прийоми та інструменти методології* – назва цього елемента говорить сама за себе – цей елемент описує прийоми та інструменти, які застосовуються в методології. Інструментами методології можуть бути складними і простими: як дошки та діаграми, так і програми, які можна застосувати для автоматизації певних завдань при розробці, що робить процес реалізації приємним. Це і є головною причиною, чому організації витрачають гроші на їх придбання та навчання своїх підлеглих як користуватися цими інструментами. Але так роблять далеко не всі, так як з розвитком технологій таких інструментів стає більше і необхідно додатково закуповувати інструменти та навчати персонал, тому частина фахівців-практиків використовують інструменти з відкритим кодом, таким чином економлять можливі витрати. Кожна методологія має власні методики, які можуть бути доречними або не актуальними для даної проблеми;

7) *автор методики* – даний елемент визначає практичні знання автора, які відповідають на питання з якою метою була створена методологія. Етап аналізу, коли всі практики об’єднані, а подібні практики визначені за різними методологіями узагальнено у таблиці 5 і яка класифікує їх за цифрами 1, 2, 3, 4 та 5:

– "1" - представляє практику, що стосуються питань планування, таких як збір вимог. Суть у захопленні мінімальних вимог найпростішим способом та почати кодування;

– "2" - це практика, яка з точки зору задоволення нестабільних вимог забезпечує поліпшення якості;

– "3" – це практика, яка забезпечує вільну роботу в команді розробників, сприяє повноваженням у прийнятті рішень, ефективному спілкуванню між собою та розвитку команди;

– "4" – представляє практику, яка забезпечує швидкий випуск товару;

– "5" - це практика, яка забезпечує постійне покращення продукту до розгортання та гнучку властивість забезпечення якості.

Після такої методології оцінки розробники можуть вибрати та підлаштувати певні практики до свого середовища відповідно до актуальності та пріоритетів проекту та замовника.

Таблиця 1.5

Визначення подібності між практиками

	Практики
XP	Процес планування(1), сорокагодинний робочий тиждень(3), невеликі випуски(2), тестова розробка(2), метафора, пріоритетність сюжетів(3), парне програмування(3), замовлення на місці(4), рефакторинг(5), колективна власність(3), простий дизайн(5) та безперервна інтеграція(5).
LD	Усунути відходи(1), максимізувати потік(2), мінімізуйте товарні запаси(1), витягніть з попиту(2), зробіть це правильно вперше(4), відповідайте вимогам замовника(2), надайте можливості працівникам(3), забороніть місцеву оптимізацію(2), співпрацюйте з постачальниками(4) та створіть культуру постійного вдосконалення(5).
Scrum	Вимоги до захоплення як відставання продукту (1), зустрічі Scrum (3), тридцятиденний спринт без змін під час спринту(2), самоорганізуючих команд (3) та зустрічі планування спринту(4).

1.4 Недоліки гнучких технологій та шляхи їх подолання

Розглянемо різні проблеми гнучких методологій:

1) *Точки зору при розробці ПЗ*. Розглянемо проблеми, які виникають серед різних методологій розробки ПЗ. Такі види діяльності як планування та оцінка завдань, обслуговування, тестування, кодування та розгортання входять до більшості процесів розробки ПЗ, і відрізняються між собою послідовністю реалізації кожного з етапів та рівень деталізації кожного етапу. Одні методології можуть реалізувати всі види діяльності, а інші тільки деякі з них. Також існують деякі відмінності, які визначають основні межі між методологіями розробки ПЗ: як методологія розцінює людей, які беруть участь у розробці ПЗ, і яка цінність задачі замовника, яку потрібно зробити.

2) *Гнучкий розвиток*. Розглянемо питання гнучких методологій, які виконуються своєрідно. У гнучких методологіях роль експерта з доменів досить унікальна. Фахівці з розробки ПЗ, які мають практичний досвід та велику кількість корисних знать, можна віднести до «мовчазних» знать, так як вони отримані шляхом практики і не зафіксовані ні в якій формі. Тому таку концепцію відносять до суб'єктивної. Однак, так як існує концепція довіри у методології гнучких технологій, що полягає у тому, що фахівці-практики найкраще знають як їм вчинити відповідно до свого досвіду і тому використання «мовчазних» знань продовжує бути частиною процесу розробки і відповідно відрізняє цю методологію від інших. Також є неоднозначним питання про можливість самоорганізації розробників у команди для досягнення цілей найкращим для них способом. У такому випадку роль керівника проекту є роллю ведучого, а не контролюючого. Різні дискусії на цю тему можна знайти у Highsmith (2004) [21] та Schwaber (2004) [17] та на http: //finance.groups. yahoo.com/group/agileprojectmanagement/.

3) Інше важливе питання – це *проведення легкої документації* у гнучкій методології, що є досить популярним з часів коли гнучкі технології набули популярності. Основою суперечок є те, що забезпечення якості ПЗ, технічне обслуговування, планування, розгортання та навчання користувачів у традиційні технології завжди пов'язували з документацією, у той час коли фахівці у гнучкій сфері наполягають, що документація має бути максимально мінімальною через те,

що правильно написаний код – це все є достатня документація і в тому числі з пов’язаними з документацією витратами.

4) Відкритим залишається питання про *рухливість* – що буде відбуватися з питаннями інноваційного мислення, який вбудований в agile розвиток, зважаючи на процеси, які стають все більш зрілими та якісними. Чи настане час, коли рухливість піде і перестане бути рухливою? Інше питання, яка набуло активного розголосу – це скільки має коштувати проект, який розробляється за допомогою гнучких процесів. Проблема полягає в оцінці вартості проекту, який заснований на ітераціях. Для того аби вирішити дане питання було зібрано декілька великих конференцій: конференції Agile Development, які проводяться в Європі та США один раз на рік.

Гнучкі технології стають все складнішими з кожним роком, так як вони проникають у інші області, такі як повторне використання ПЗ, архітектура підприємства тощо, що ускладнює обробку ПЗ. Але якщо цього не помічають прихильники даної методології, можливо, через деякий час можна буде спостерігати картину революції в розробці ПЗ, яка виникне для збереження спадщини рухливості.

Майбутні тенденції гнучкої розробки ПЗ. Чому гнучка розробка стає все більш популярною і набирає обертів з кожним роком? По перше, це пов’язано з тим, що про гнучку методології є достатньо описової та аналітичної інформації. По друге, гнучка методологія цікавить науковців, які збирають та аналізують дані для спростування та доведення різних анекдотичних даних про гнучкі процеси. По третє, фахівці-практики не перестають постійно обмінюватися свої досвідом. В загальному все це спрямовує гнучкі технології на все більший попит у великій кількості організацій на консультації та тренінги на тему гнучких технологій. Poppendieck зазначив, що більший попит є в Північній Америці, Європі та Японії, де його книга про розробку ПЗ продана понад 10 тисяч примірників. Іншою важливою подією у світі гнучких технологій є їх пропозиція спонсорувати гнучкі дослідження, що звісно допоможе у розвитку гнучкої методології.

ВИСНОВКИ ДО РОЗДІЛУ 1

У першому розділі було представлено огляд гнучких методологій, де основна увага приділялася не опису кожної існуючої гнучкої методології, а огляду методологій, які могли надати досить повне визначення гнучкості та гнучкого забезпечення якості. Дане визначення представлено з одного боку як теоретичне, а з іншого – як практичне, що може бути корисним фахівцям, які і філософськи відносяться до даної методології і для фахівців, які розвиваються і практикують регулярно гнучку методологію. Також була представлена методологія оцінювання для кращого розуміння гнучких процесів та розкрити основні принципи діяльності різних процесів цієї методології. Метою цієї методології є досягнення балансу між дотримуванням тільки однієї методології і триматися за всі одночасно, тобто поглянути на всі методології із загальної точки зору і використовувати тільки краще і необхідне у кожній з них. Також були розглянуті недоліки гнучкої методології розробки програмного забезпечення та шляхи подолання проблем, які вже використовуються на практиці.

РОЗДІЛ 2

АРХІТЕКТУРА ПРОГРАМНОЇ СИСТЕМИ В ГНУЧКИХ ТЕХНОЛОГІЯХ

При обговоренні архітектури програмної системі, частини якої можуть змінюватися, у випадках коли розробляються подібні системи з різними сценаріями використання, важливу роль займає змінюваність – очікувані і попередньо заплановані зміни. Змінюваність включена та відображається в архітектурі ПЗ у великій кількості сучасних систем, наприклад, само-адаптивні системи, системи, які підтримують динамічні веб-сервіси тощо. Тобто під час проектування архітектури ПЗ необхідно пам’ятати про керування змінюваністю – процеси, які стосуються визначенням (ідентифікація, де потрібна змінюваність), обмеженням, впровадженням та керуванням змінюваністю.

Важливою є гнучка парадигма – гнучкі методи, які створені на основі принципів маніфесту, та практик, які забезпечують адаптацію до змін. Змінюваність зручно використовувати у випадках проектування систем, коли потреби клієнтів або контексти недостатньо відомі, а гнучку розробку – з будови програмних рішень ще до того, як продукт буде цілковито зрозумілим.

Тож за допомогою поєднання при проектуванні архітектури гнучкої парадигми та змінюваності можливо досягти гнучкої адаптивної програмної архітектури, і відповідно системи. Такі зміни забезпечують доступність інформації про змінність на різних етапах розробки – при тестування та реалізації системи. Тому такий підхід поєднання може бути корисний проектам з часто змінюваними вимогами, важливістю постійного зворотного зв’язку з клієнтами, короткими термінами та графіками, необхідністю зосередитися у більшій мірі на розробці ПЗ, а не на документації та плануванні, що дозволить керувати змінюваністю за короткий проміжок часу і з невеликої кількістю зусиль.

Інша частина цього розділу розглядатиме виникаючу архітектуру, як і в якій мірі претензії на виникаючу архітектуру є істинними. Тому спочатку буде розглянута мета, діяльність і завдання архітектурної роботи. Згодом буде проаналізовано, чи підходить виникаюча архітектура для заміни різних видів діяльності та цілей архітектурної роботи, зважаючи на її мету. Під час обговорення, сильні та слабкі сторони явних і виникаючих архітектурних робіт будуть протиставлятися одна одній, і буде запропонований спільний підхід.

2.1 Змінюваність програмної системи в процесі проектування

Здібність продукту ПЗ адаптуватися до заздалегідь конкретних вимог у спланованому вигляді називають змінюваністю. Термін змінюваності несе в собі розуміння того, що зміна відбувається планово, не через певні проблеми в системі, непередбачувані потреби користувачів або технічне обслуговування.

На момент проектування певні частини архітектури та системи можуть бути не достатньо визначеними або залишатися змінюваними, саме змінюваність і визначає їх, дозволяючи розробляти різні частини системи або архітектури. Яким же чином визначена змінюваність в архітектурі ПЗ? Вона визначена як точки зміни – місця в архітектурі, де фактично може статися певна зміна, та варіанти – фактичні рішення для цих точок змін. Тому при проектуванні необхідно максимально забезпечити розуміння того, що таке змінюваність та як вона буде впливати на архітектуру ПЗ, так як змінюваність відбувається на різних етапах ЖЦ ПЗ. Відповідно при керуванні змінюваністю необхідне точне розуміння змінюваності артефактів ПЗ за весь час ЖЦ продукту.

Термін керування змінюваністю не тотожний терміну управління нею, так як управління(управління залежностями між змінними, підтримання та постійну популяцію варіативних опцій з новими змінами, видалення опцій, розповсюдження нових змін тощо) змінюваністю є тільки частиною керування(в тому числі ідентифікація змінюваності, представлення і її реалізація тощо) за словами Svahnbergetal.

2.2 Змінюваність та гнучкість архітектури в гнучких технологіях

Також важливою є гнучка парадигма, яка забезпечує адаптацію та зміни – гнучкі методи, які створені на основі принципів маніфесту, та практик, які використовують ці самі принципи адаптації та змін у певному проекті. Всі гнучкі методи реалізують подібні принципи та цінності та забезпечують ітеративні додаткові ЖЦ. Гнучка парадигма створена для приймання та пристосування до змін, але аж ніяк до строгих вимог до проектування архітектури ПЗ. Також необхідно нагадати, що гнучкі методи забезпечують безперервне постачання та забезпечення результатом своїх клієнтів завдяки принципу зворотного зв'язку. Але окрім загальних принципів, які закладені у кожен гнучкий метод, у кожному з них є своя певна практика, яка відрізняє певний метод від інших – наприклад, парне програмування в екстремальному програмуванні XP, спринт у Scrum тощо.

Гнучка парадигма базується на маніфесті Agile, який включає в себе 12 принципів, вказаних у табл. 2.1.

Таблиця 2.1

Принципи маніфесту Agile

Принцип	Опис
1	Найвищий пріоритет базується на тому, аби задовольнити користувачів за допомогою раннього і безперервного постачання ПЗ.
2	Змінюваність вимог використовується для задоволення клієнта і її можна використовувати навіть на пізніх етапах процесу розробки.
3	Релізи ПЗ мають випускатися часто і з коротким терміном доставки.
4	Замовник та розробники ПЗ мають постійно взаємодіяти між собою та працювати разом протягом всього проекту.
5	Проекти мають бути побудовані у колі мотивованих людей і в середовищі, яке забезпечує їх потреби та сприяє довірі у певних рішеннях.

Продовження таблиці 2.1

6	Надається перевага комунікації між групами розробників «віч-на-віч», тобто особиста передача цінної інформації.
7	На основі робочого ПЗ вимірюється прогрес.
8	Гнучкі процеси сприяють постійному темпу сталої розробки на невизначений термін, і користувачі, розробники та спонсори мають підтримувати даний напрям.
9	Постійна увага приділяється технічній досконалості, що в результаті підвищує гнучкість системи.
10	Збільшення кількості робіт, які є недовиконаними.
11	«Самоорганізація» команд є важливою частиною для забезпечення кращої архітектури, вимог та проектів.
12	Команди розробки ПЗ мають постійно вдосконалюватися та намагатися ставати все більш ефективними, відповідно пристосовуючи свою поведінку до розвитку.

Якщо не зважати увагу на початкову природу змінюваності та гнучкої парадигми, можна побачити, що між цими термінами є певних зв’язок та певні загальні для них принципи, які наведені нижче:

– Обидва поняття керування змінюваністю та гнучкої парадигми тим чи іншим чином пов’язані з постійно змінюваними та адаптивними вимогами: змінюваність очікує змін, а гнучкі методи обробляють їх. Все це дозволяє працювати з очікуваними та непередбачуваними змінами, так як певні непередбачувані зміни можуть співпадати з точками змін, відповідно дозволяючи без проблем вносити зміни в архітектуру. Таким чином завчасне рішення про поєднання гнучкої парадигми та змінюваності на етапі проектування архітектури забезпечує можливість вносити в неї зміни та додавати нові варіанти, у той час як в архітектурі, де це передбачено не було, зробити це набагато важче.

– Змінюваність та гнучка парадигма вимагають зворотного зв’язку від користувачів.

– Змінюваність та гнучка парадигма працюють в рамках певної області. Ступінь змін виявляє діапазон адаптації.

– Гнучкість та змінюваність направлені на мінімізації роботи, яка має бути виконана. Таким чином, гнучкість відкладає роботу, поки вона буде необхідна, а змінюваність постійно прогнозує, що буде необхідно (точки зміни) та створює певні задачі (варіанти) для вирішення певних цілец.

2.3 Поєднання Agile парадигми з управлінням змінюваністю

Змінюваність зручно використовувати у випадках проектування систем, коли потреби клієнтів або контексти недостатньо відомі, а гнучку розробку – з будови програмних рішень ще до того, як продукт буде цілковито зрозумілим. Відповідно виникає бажання поєднати управління змінюваністю та гнучку парадигму, тобто використовувати гнучку парадигму в обробці змінюваності на рівні архітектури ПЗ. Таким чином можна сформулювати переваги даного поєднання, що описано в наступних трьох пунктах:

1) Гнучка парадигма забезпечує легше управління змінюваністю, так як гнучкі процеси розробки ПЗ, які покладені у гнучку парадигму, вимагають зменшення накладних витрат.

2) Завдяки принципу зворотного зв’язку гнучкої парадигми, про певні зміни архітектури або ПЗ можливо миттєво дізнатися у клієнтів.

3) І наостанок – гнучка парадигма дозволяє додавати нові точки змін та варіанти змін в архітектуру, що полегшує майбутню еволюцію змінюваності.

Але об’єднання змінюваності та гнучкої парадигми, на жаль, обмежена. Більшість практичних робіт з використанням даної практики було проведено у контексті продуктів гнучкої інженерії, де змінюваність була ключовою концепцією – наприклад, Перший міжнародний семінар, який був проведений у 2006 році, з лінії продуктів гнучкої інженерії, який став першим внеском та досвідом про можливість

поєднання змінюваності та гнучкості. Також пізніше були в літературі опубліковані огляди лінії продуктів гнучкої архітектури, в яких були перелічені причини такого поєднання (зниження витрат через нестабільні ситуації у бізнесі), методи які використовувалися та майбутній можливий розвиток (еволюція змінюваності). Але є роботи які свідчать про те, що інтеграція попереднього планування продукту та гнучкої розробки є досить важкою, тому вони мають бути пристосовані таким чином, аби вони могли зберегти свої основні характеристики і водночас не ускладнити інтеграцію програмного продукту.

Було представлено і інше, але вже промислове дослідження з гнучкої інженерії Hansenand і Faegri [12], яке показало, що впровадження гнучких продуктів є довготривалим процесом, так як включає в себе велику кількість інших процесів – управління знаннями, планування продукту, інновації, організаційні процеси тощо. Похялайнен представив інший досвід, який включає легку функціональну модель [9]. Інші приклади об’єднання інженерії продуктової лінії та гнучкої парадигми використовують легкі методи моделюванні функцій [14], як у Похялайнен. Важливим внеском у еволюцію змінюваності вніс Ghanametal, який виявив вимоги змінюваності за допомогою використання тестів з прийому і вніс пропозицію про керування реактивною змінюваністю. Він зміг довести, що поєднання гнучкої парадигми та інженерії продуктів можливе, але процес планування гальмує розвиток та гнучкість процесу розробки ПЗ.

Але, на жаль, за межами сфери продуктових ліній не вдається знайти багато прикладів по об'єднанню змінюваності та гнучкості, так як архітектурі ПЗ при дослідженні цього питання не було приділено достатньо уваги.

Тому одним з важливих питань, які розглядаємо у цій главі – це яким чином можна використовувати змінюваність для майбутнього спрощення розробки ПЗ. Одним із запропонованих підходів є динамічні архітектури, які дозволяють під час виконання забезпечити адаптивність системи. Прикладами можуть бути дослідники динамічних виробничих ліній Шокрі та Алі Бабар [15] або Халлштайсен; а також динамічних реконфігурацій сервісно-орієнтованих архітектор Фідейро і Лопес; також був запропоноване самоадаптивне ПЗ Oreizyetal. [12]

2.4 Проблеми управління змінюваністю

При об’єднанні змінюваності та гнучкої технології для розробки програмної архітектури існують декілька проблем:

– Необхідно провести комплексний аналіз архітектури для виявлення точок змін, в тому числі і джерела варіації та варіанти – фактичні рішення для цих точок змін. Але так як керування змінюваністю вимагає попереднього аналізу та передбачення змін – це конфліктує з тим, що гнучка розробка зосереджена в основному на розробці ПЗ, яке задовольняє клієнта, що включає зменшення витрат та інвестиції.

– Наскільки відомо, гнучка розробка за рахунок мінімізації витрат намагається відстоювати думку про легку документацію, протестуючи проти надмірної. А документація змінюваності навпаки вимагає документування як частину процесу передачі знань між практиками аби архітектура могла повною мірою відображати змінюваність, тому проектувальник має документувати максимально детально [11], що суперечить гнучкій розробці.

– Важливою концепцією гнучкої розробки є часті та швидкі випуски продукту, що відповідно задовольняє потреби клієнта. В даній ситуації виникає конфлікт зі змінюваністю, так як перед тим як випустити певний реліз продукту, необхідно проаналізувати детально змінність аби пізніше інфраструктура була гнучкою і змінюваною, після цього налаштувати її та після цього вже доставити продукт клієнту [14]. Все це ускладнює і суперечить принципу простоти гнучкої парадигми, яка забезпечує не майбутні, а існуючі вимоги, і не переймається за різні точки зміни в архітектурі, які існують поза поточною ітерацією [11].

– Коли відбуваються зміни у вимогах гнучкі методи швидко відповідають на них короткими спринтами, що вимагає протягом всього ЖЦ ПЗ тримати зворотній зв’язок із клієнтами [11]. Комунікація є також важливої і на рівні архітектури ПЗ при керуванні змінюваністю, але на цьому рівні вона відбувається не так активно.

– Для гнучких методів та для керування змінюваністю використовуються різної складності інструменти. Наприклад, для гнучкої парадигми вони прості та ефективні (веб-інструменти для розробки на основі тестів, бумажні історії тощо), а для керування змінюваністю досить складні інструменти (наприклад, Gearsa або Pure) [11].

2.5 Ідея виникаючої архітектури

Останнім часом популярною темою в гнучкій методології є тема виникаючої архітектури, яка тісно пов'язана з суперечками, чи архітектор є важливою і необхідною частиною в гнучких проектах або всі архітектурні роботи можуть бути виконані командою розробників. Але перед тим як проаналізувати це твердження, необхідно розуміти, що саме означає термін виникаюча архітектура. Для цього спочатку наведемо визначення поняття «виникнення».

2.5.1 Коротке визначення «виникнення» та ідея виникаючої архітектури

Джеррі Гольдштейн пропонує таке визначення для цього терміну: «Виникнення – це є поява нових і когерентних структур, шаблонів і властивостей під час самоорганізації в складних системах» [14]; Вікіпедія ж тарктує, що «... виникнення - це те, як складні системи і структури (патерни) виникають з множинності відносно простих взаємодій» [14].

Існує безліч інших подібних трактувань, але вище згаданих достатньо для опису основних характеристик (рис. 2.1), які є найбільш актуальними в контексті виникаючої архітектури:

– *комплексні системи* – виникнення виступає в складних середовищах;

– *структури, шаблони та властивості* – виникнення будує нові структури, шаблони та властивості;

– *становлення* – це найважливіша характеристика виникнення. Поява нових структур і шаблонів не є головним, а відбувається як побічний ефект виконання, начебто, непов'язаного з ними завдання час від часу;

– *відносно прості взаємодії* – передбачалося що не пов'язані між собою завдання, принаймні, є на порядок простіше, ніж виникаючий результат;

– *самоорганізація* – зважаючи на теорію систем, самоорганізація завжди має мету (завдання, яке потрібно вирішити, або досягти бажаного кінцевого стану) і оточений набором обмежень.

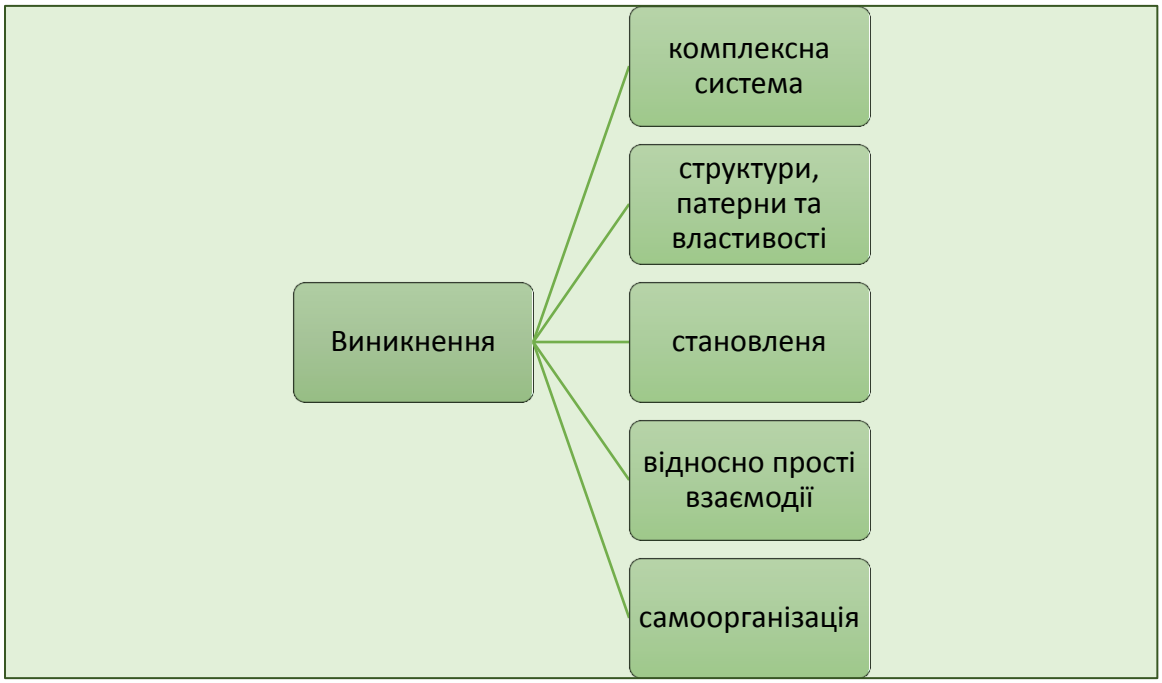

Рис. 2.1. Основні характристики виникнення

У не зовсім формальному відношенні люди часто використовують виникнення, якщо вони висловлюють певні розбіжності, на приклад: "Результат більше, ніж сума його частин", або "Один плюс один більше двух".

Після того, як ми з'ясували, що таке виникнення і перерахували його характеристики, зауважимо як він може бути застосований для визначення виникаючої архітектури. Метою виникаючої архітектури є створення архітектури для нетривіальної системи – її фундаментальні структури, шаблони (патерни) і властивості. Взаємодії, які запропоновані для досягнення цієї мети, є заходами, які

розробники, в основоному, виконують регулярно: бізнес-властивості втілюються в безперервному циклі реалізації та рефакторингу. Зазвичай цей цикл поширюється до цикла повної розробки, який керується тестами:

- напишіть тест, який не виходить;
- запишіть код, щоб сам тест пройшов успішно;
- здійснити рефакторинг коду, не пошкодити тест;
- продовжити з подальшим тестом.

До того ж, декілька керівних принципів SOLID [14] застосовуються для забезпечення напрямку рефакторингу, який сприяє створенню більш вдалих об'єктно-орієнтованих (ОО) проектів. Потрібно звернути увагу на те, що принципи SOLID є основою проектування, а не архітектури. Вони використовуються для деталізації рівня ПЗ. Цей процес виходить чіткішим, ніж з використанням звичайної архітектури.

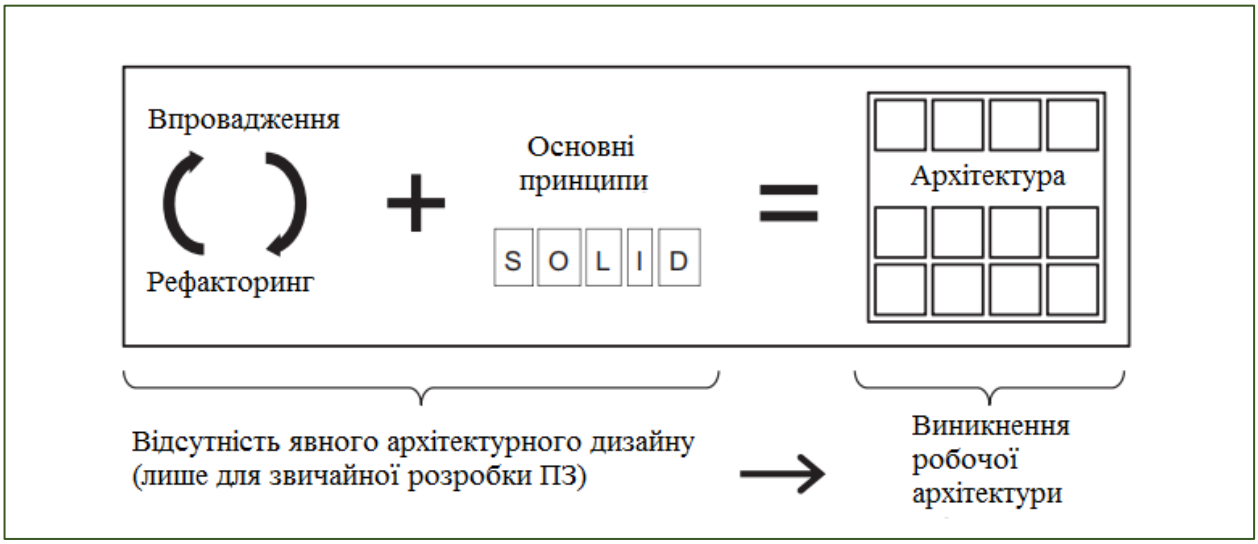

Рис. 2.2. Вимоги виникаючої архітектури

Концепція виникаючої архітектури полягає в тому, що виконання циклу реалізації і рефакторинга все заново, застосовуючи деякі принципи проектування, а точніше принципи SOLID для керування рефакторингом, приведуть до досконалої архітектури без будь-яких явних архітектурних робіт (див. рис. 2.2) .

Заступники виникаючої архітектури також доводять, що отримана архітектура є не тільки досконалою, але й найбільш оптимальною, пропонуючи наступні міркування: рефакторинг завжди є спрямованим на створення найпростішого і найменшого рішення, яке працює і виконує керівні принципи. Більша частина прихильників гнучкої методології йдуть від цієї пропозиції вперед і доводять, що архітектори взагалі вже не потрібні, тому, що лише група розробників зможе створити архітектуру, виконуючи свою звичайну роботу з розробки ПЗ.

Проте, вказівки автора в конкретних комерційних проектах не підтвердили змальовану раніше вимогу. Згодом, деякі проекти, що підходять до чистого виникаючого архітектурного підходу, страждали від проблем, які майже постійно оглядаються архітектурними роботами. Постійно відсутні зрозумілі якісні абстракції, що впливає на зниження розуміння всієї системи та зменшення здатності до трансформації. Поза тим, окремі обов'язки не були розділені таким чином, щоб також призвести до зменшення здатності трансформації. Реалізуючи один проект, команда розробників не наважилася впровадити певні зміни до бізнес-логіки в системі, яку вони тільки но почали створювати з нуля 5 місяців тому, бо це було дуже екстримально.

2.5.2 Мета, діяльність та цілі архітектури

Для того, аби проаналізувати, чи є змога виникаючій архітектурі замінити явну архітектурну роботу, необхідно, перш за все, мати глибоке розуміння мети, діяльності та цілей архітектурної роботи. Проблема більшої частини існуючих трактувань полягає в тому, що вони або не є повними, і зазвичай, зосереджуються лише на певних цілях, або взагалі є розмитими. Виходячи з цього, є неможливим випробування (рішення задачі) виникаючої архітектури, якщо вона підходить під це визначення. Через те, в цьому розділі застосовується інакший підхід. Він спробує відповісти на подальші запитання (рис. 2.3):

– Чому? – Чому архітектурна робота загалом відбувається? Яка її мета? Для чого це є добрим?

– Як?– Як відбувається архітектурна робота? Які заходи характерні ґрунтовним архітектурним роботам?

– Що? – Які цілі архітектури? Які важливі частини?

Відповіді на дані запитання базуються на досвіді автора цієї статті. Поміж того, вони є результатом довготривалих багаторічних спостережень в комерційних проектах, обговорених з багатьма групами IT-спеціалістів, та перш за все, з архітекторами, і є резуьтатом тестування відповідей на варіаційні описи в сучасній літературі.

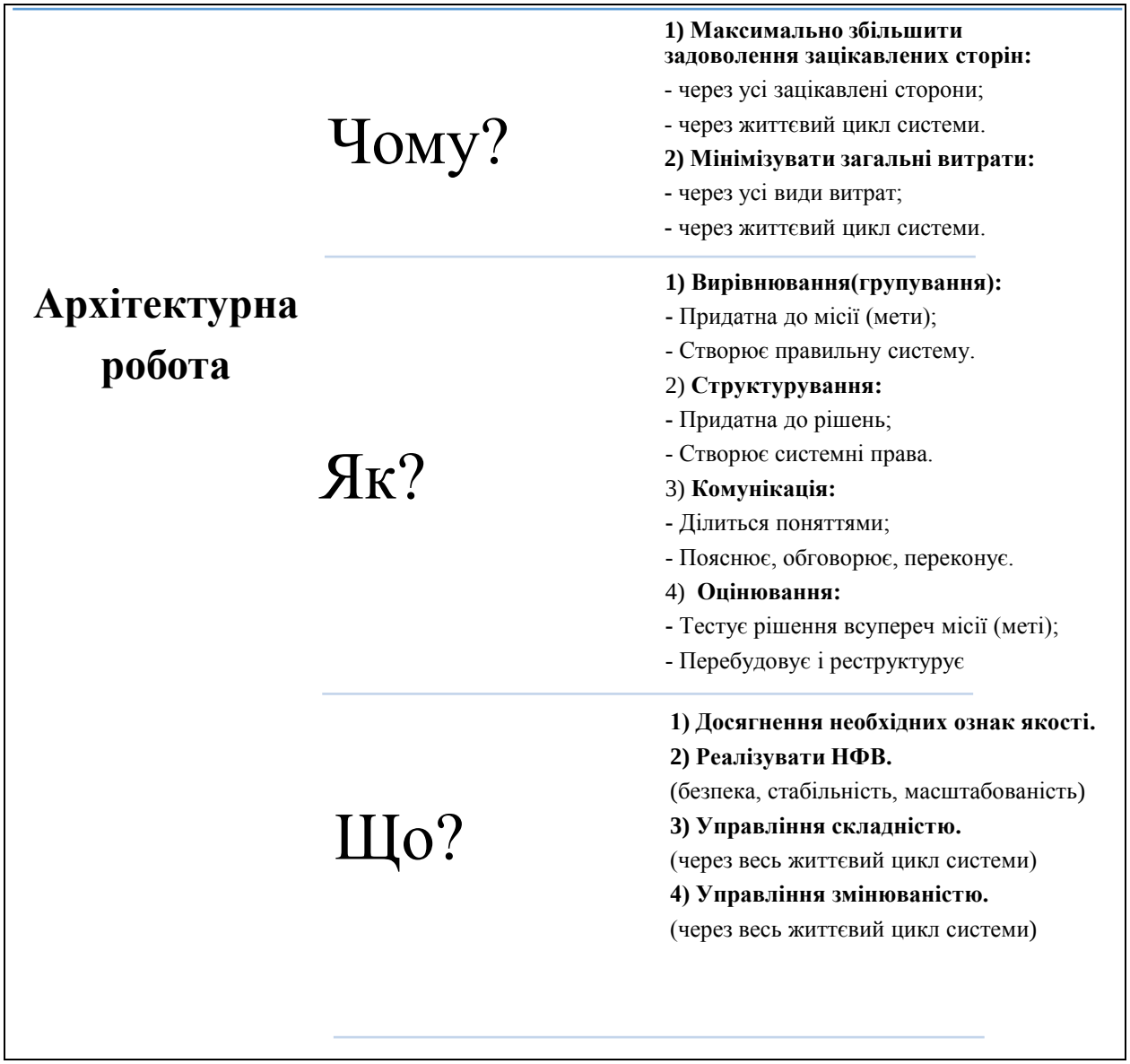

Рис. 2.3. Питання виникаючої архітектури

2.5.2.1 Мета архітектури

Загалом всі трактування архітектури приходять до запитання «Що?». А отже, емпіричний підхід застосовується для того, аби відповісти на запит архітектури: коли запитання "чому?" досить часто з’являється в комерційному проекті, то воно зводиться до подальших двох мотивів, практично без залежності від конкретної діяльності:

- максимізуйте задоволення когось;
- звести до мінімуму витрати на щось.

Архітектурі необхідним є врахування цих двох мотивацій. Характерною рисою архітектури для багатьох суміжних видів діяльності є те, що дві згадані вище цілі мають більше одного виміру:

- архітектура має за мету максимальне задоволення всіх зацікавлених сторін. Мало буде просто задовольнити одну із сторін – як приклад, розробників. Бажано також проаналізувати операційний відділ, керівників проектів, команди підтримки і таке інше. Зазвичай сторонами зацікавлених сторін виступають розробники, користувачі, клієнти, керівники, менеджери проектів, менеджери розгортання, операцій, управління інфраструктурою, співробітники служби безпеки, підтримка, персонал гарячої лінії та багато інших. Без сумніву, є неможливим максимально задовольнити кожного індивідуума, але необхідно шукати збалансований компроміс, який зможе максимізувати загальне задоволення всіх членів;
- архітектура має за мету мінімізувати загальні витрати, і в результаті задоволення притягнутих сторін, потрібно продивитися всі види витрат і не зосереджувати увагу тільки на одному з них. Типові типи витрат – це розробка, обладнання, ліцензія, розгортання, операції, потужність, підтримка, обслуговування, використання тощо. Зауважимо, що неможливо мінімізувати кожен тип витрат окремо, але загальні (це сума різних типів витрат) повинні бути добре оптимізованими;

– час є другим виміром архітектури, який має бути взятий до уваги, також він є недостатнім для максимального задоволення та мінімізації витрат протягом обмежених часових рамок (зазвичай, проекту). Замість того необхідно розглянути увесь, що залишився ЖЦ системи, на яку був здійснений вплив. Приведемо приклад: відносно легко звести до мінімуму витрати на розробку проекту, але, як правило, майбутні проекти і команда з технічного обслуговування повинні будуть дорого заплатити, якщо вони не будуть враховані заздалегідь, оскільки, немає се[48] створювати більш адаптованого рішення для системи, яка близька до кінця його ЖЦ.

Тепер можемо зробити висновок, що мета архітектури полягає в максимальному задоволенні зацікавлених сторін і мінімізації загальних витрат протягом усього життєвого циклу постраждалої системи.

2.5.2.2 Діяльність архітектури

Через те, що більшість трактувань архітектури підходять тільки для архітектури, змішаний підхід застосовується для відповідей на запитання. Щоб перевірити повноту визначених заходів, вже існуючі моделі процесів архітектури, такі як для arc42 [4], використовують для перевірки аби жодна з дій не була охоплена типами діяльності, виявленими раніше. Застосовуючи цей підхід, було визначено наступні чотири види діяльності:

1) *Вирівнювання (групування)* – коли всі дії пов'язані з забезпеченням аби рішення відповідало його місії. Це полягає в тому, щоб розмовляти із зацікавленими сторонами, з'ясовувати їхні потреби та вимоги, укладати компроміси, вирішувати суперечності у вимогах та багато іншого. Враховуючи спостереження автора, на діяльність такого типу часто не звертають увагу менш досвідчені архітектори.

2) *Структурування* – коли всі види діяльності пов'язані із створенням дизайну рішення. Воно складається з прийомом інформації, яка була зібрана в процесі вирівнювання, та перевтілення інновації в розробку рішення. Простіше кажучи, структурування полягає в тому, щоб всі дії мали свою мету і робити все

правильно. Автор дає змогу зрозуміти, що мало досвідчені архітектори зосереджують свою увагу лише на діяльності конкретного типу.

3) *Спілкування* – коли «просування» проекту рішення є важливою, але часто недооціненою, складовою архітектурної роботи. Зрозуміти це допомогає пояснення концепцій і ідей у дизайні, та прийнятих рішеннях із розумінням чому саме вони були прийняті, переконуючи людей забезпечити реалізацію дизайну і таке інше.

4) *Оцінка* – коли архітектурні проекти вже мають бути підтвердженими, аби побачити відповідність їх місії. Вимоги змінюються поступово, з'являються нові потреби і існуюча архітектура може більше не підтримувати їх. Отже, необхідно регулярно оцінювати архітектуру системи її придатності та виводити необхідні підтримки для відновлення рішення своїми призначеннями.

Можемо далі розрізняти типи діяльності, але даної деталізації достатньо щоб дати відповідь на запитання про те, чи може виникаюча архітектура замінити явну архітектурну роботу. Додаткові спостереження дозволяють проігнорувати два з чотирьох типів діяльності для решти розділу:

– Комунікація необхідна завжди: структура, що здається очевидною для одного індивідуума, може сприйматися по-іншому другим, і часто дизайнерські рішення не є очевидними. Отже, архітектура завжди повинна підтримувати зв’язок незалежно від того, чи вона була розроблена явно або виникла.

– Оцінювання можна пояснювати як підмножину групування в даному контексті: діяльність, пов'язана з оцінками, подібна до тих, що стосуються узгодження, але оцінка зазвичай має набагато більш обмежений обсяг. Отже, достатньо оскаржити виникаючу архітектуру з узгодженням діяльності.

Розглянемо атрибути якості:

– *Зручність використання* – це певна кількість атрибутів, які покладаються на зусилля, необхідні для використання, на індивідуальну оцінку даного використання заявленим, або передбачуваним набором користувачів: зрозумілість, можливість навчання, керованість, привабливість, дотримання зручності використання.

– *Ефективність* – це ряд ознак, які спираються на відносини між рівнем виконання та продуктивністю програмного забезпечення та сумою ресурсів, що використовуються під час встановлених умов: поведінка у часі, використання ресурсів, дотримання ефективності.

– *Супроводжуваність* – це ряд ознак, які спираються на поліпшення, які можуть бути зроблені під час певних модифікацій: аналізованість, змінюваність, стабільність, тестованість, дотримання супроводжуваності.

– *Мобільність* – це ряд ознак, які спираються на переміщення програмного забезпечення, що може функціонувати у різних середовищах: адаптованість, можливість встановлення, співіснування, замінюваність, дотримання мобільності.

Атрибути якості в областях функціональності, надійності, зручності використання, ефективності та мобільності говорять про властивості системи, які можуть бути реалізованими (впровадженими) як звичайні функціональні вимоги. Вимога повинна бути зрозумілою: рішення розроблено (спроектовано), впроваджено (або налаштовано) і поставлено – із залученням, або без залучення циклів зворотного зв'язку та уточнень і поліпшень. У гнучких підходах вимоги можуть навіть бути запам’ятовані як історії користувачів. Існує лише одна різниця від функціональних вимог – це те, як вимагаються експертні знання в різних областях (наприклад, домен безпеки).

Атрибути в області супроводжуваності, з іншого боку, потребують принципово іншого підходу. Вони звертаються до управління складністю та змінами, що відповідає визначенню архітектури, яке придумав Джон Закман: «Причини, по яких необхідна архітектура: складність і зміни» . Відображення цих атрибутів якості в дії по проектуванню призводить до проектування зрозумілості (керує складністю) та проектування змін (керує змінами).

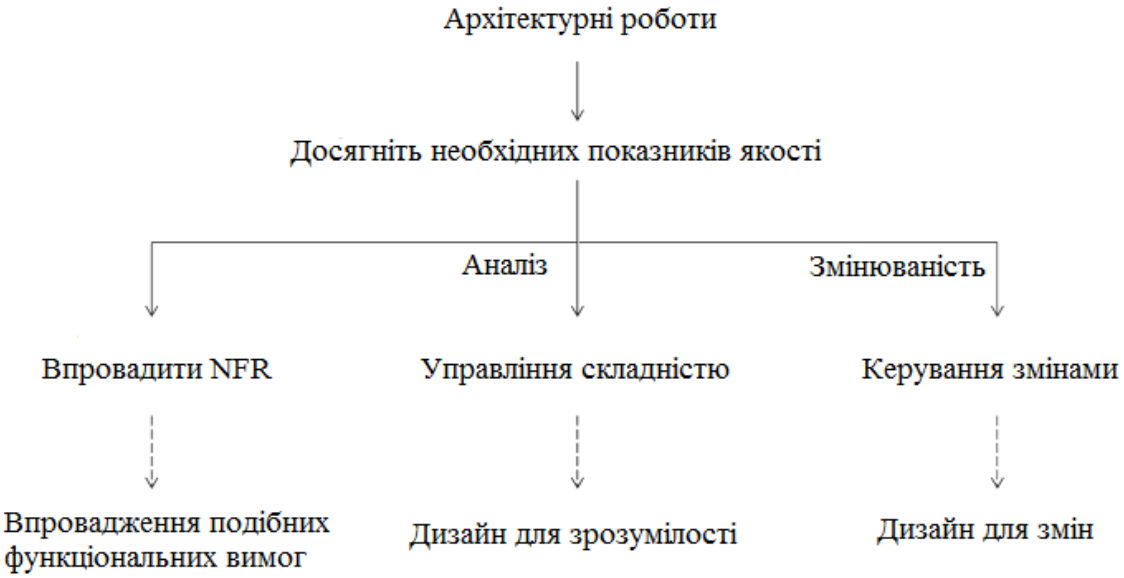

Рис. 2.4. Цілі, які можна отримати з атрибутів якості ПЗ

2.6 Аналіз виникаючої архітектури

Дослідивши мету, дії і цілі архітектури, наступним кроком має бути аналіз того, чи можуть описані дії бути заміненими архітектурою виникнення з відповідальністю до мети архітектури взагалі.

2.6.1 Вирівнювання

Вирівнювання – це частина архітектурної роботи, де архітектор гарантує, що рішення відповідає своїй місії (робить правильні речі). Загалом воно складається із багатьох розмов з зацікавленими сторонами, намагаючись зрозуміти їхні потреби та рушійні сили, щоб переконатися в тому, що вимоги правильно розуміються, що всі важливі вимоги розглянуті, що конфлікти та спірні вимоги вирішені, і за допомогою цього, знання поширюються в процесі реалізації серед команди. Цей вид роботи вимагає певного набору навичок (див. [8,9] для більш розгорнутої інформації):

– Взаємодія із залученими сторонами шляхом (про-)активного спілкування, переговори, посередництво, робота в команді, врегулювання конфліктів, знаходження компромісів, слухання, розмова, переконання, і т. д.

– Розуміння «мов» зацікавлених сторін для розуміння їх мотивацій, потреб та вимог.

– Розмовляти на мовах зацікавлених сторін, щоб стати прийнятими, як професійні колеги. Зацікавлені сторони часто не сприймають людей, які не «розмовляють їхньою мовою», що робить більш складним спілкування та зменшує готовність зацікавлених сторін обговорювати свої проблеми та вимоги.

– Розуміння навколишньої організації та політик, так щоб не потрапити під перехресний вогонь.

Вирівнювання є вирішальним фактором успіху щодо цілей архітектурної роботи. Розглянемо два спостереження щодо виникаючої архітектури:

1) Вирівнювання не виникаюча робота, а явна. Вона не виходить з повторного циклу реалізації та рефакторингу.

2) Набір навичок, що необхідний для вирівнювання, не є набором навичок розробника. Вони сильно відрізняється від набору навичок, необхідних для реалізації ПЗ, також для його створення потрібно багато часу і досвіду. Зазначимо, що ці навички неможливо придбати лише за допомогою реалізації ПЗ.

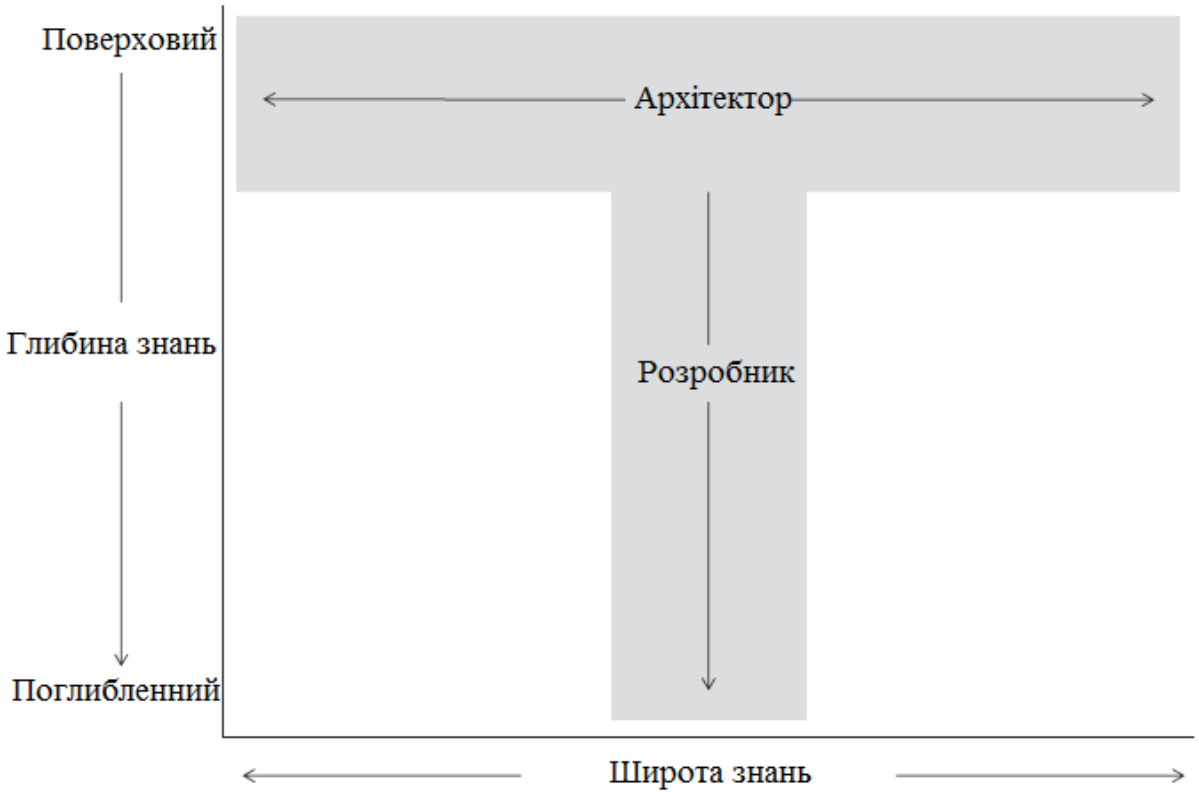

Рис. 2.5. Різні набори навичок для архітекторів та розробників

За допомогою схеми (див. рис. 2.5) зможемо проаналізувати останнє спостереження більш детально. Ми бачимо спрощений розподіл знань для різних ролей. На осі X показана широта знань, а на осі Y – глибина знань. В ролі розробника ПЗ потрібне дуже глибоке розуміння в досить обмеженій області: мова програмування, фреймворки, бібліотеки, шаблони проектування (патерни), методи кодування тощо.

З іншої сторони, архітектурна робота (а особливо вирівнювання) вимагає дуже великого досвіду: знання областей зацікавлених сторін для того, щоб зрозуміти їхні потреби, вимоги і вміти спілкуватися на однаковому рівні. Важливо і мати багато програмних навичок.

За допомогою цього спостереження ми можемо зробити висновок: концепція про те, що гнучка команда розробників може повністю замінити архітектора, є сумнівною, тому що загалом не можна вважати, що команда розробників має необхідний набір навичок для виконання роботи з вирівнювання.

Можемо також говорити про те, що все ще залишається альтернатива взагалі не виконувати вирівнювання і покладатися на чисто виникаючу архітектуру для отримання необхідних результатів без чіткої роботи з вирівнювання. Таким чином, зробимо зауваження щодо такого підходу:

1) Уважно не розмовляючи із зацікавленими сторонами і активно не вирішуючи конфлікти, існує дуже високий ризик того, що в свою чергу і архітектурні вимоги не будуть правильно зрозумілі та не будуть реалізовані. Навіть якщо це може стати видимим у більш пізньому циклі зворотного зв'язку, це порушить мету архітектурної роботи: максимальне задоволення зацікавлених сторін і мінімізацію загальних витрат. Крім того, наслідки неправильних архітектурних рішень не стануть видимими протягом тривалого часу, особливо якщо вони впливають на зрозумілість або змінюваність (маються на увазі невідкладні цикли ЗЗ у гнучких підходах можуть не допомогти).

2) Без вирівнювання існує високий ризик того, що потреби розробників переоцінюються, а вимоги всіх інших сторін не враховуються. Це також може порушити мету архітектури.

Підсумовуючи, можемо стверджувати, що вирівнювання є важливою частиною архітектурної роботи; його потрібно робити явно і не можна опускати або замінювати виникаючим підходом, не порушуючи загальної мети архітектурної роботи.

2.6.2 Структурування

Структурування є важливою частиною для орієнтації та спілкування (комунікації) в ролі архітектурної діяльності. Орієнтаційний аспект стверджує, що людський мозок не здатний справитись з великою кількістю інформації на одному рівні. Для подолання цього обмеження, інформація розміщується в структурах, які допомагають людям швидко знаходити необхідну частину інформації та ефективніше працювати з інформаційними частинами та відносинами між ними. Типові програмні системи складають тисячі або мільйони інформаційних деталей. Ці системи роблять організаційну структуру незамінною.

Орієнтації структура завжди підтримує конкретний варіант використання, як засіб. Потрібно пам’ятати, що структура, яка підтримує один варіант використання найкраще, може ніколи не підтримувати інший. Приведемо приклад: структурування великої кількості електронних листів за датою допомагає ефективніше знаходити повідомлення електронної пошти, отримані в певний термін. Проте, ця структура взагалі не допомагає, якщо всі електронні листи, пов'язані з конкретною темою. Взагалі, неможливо створити структуру, яка буде підтримувати всі можливі випадки використання однаково точно.

Варіанти використання, що потребують підтримки архітектурних структур, являється завданнями для архітектури: реалізація не функціональних вимог, проектування зрозумілості та проектування змінюваності. В наступних розділах

буде проаналізовано, чи допомагає виникаюча архітектура підтримувати ці варіанти (сценарії) використання.

Можемо зробити ще одне спостереження, враховуючи виникаючу архітектуру: архітектурна структура повинна і має бути пов’язана (тобто, комунікована), тому, що вона використовується як засіб комунікації. Для всіх груп зацікавлених сторін, які не є розробниками, це означає, що деякі архітектурні документи, які не є кодами, повинні бути створені, тому що код розуміють лише розробники. Створення цієї документації завжди є точною роботою. Але, то не є результатом циклу кодування та циклу рефакторингу. Отже, структурування завжди включає деяку документацію, якої не було на стадії виникнення і становлення.

2.6.3 Реалізація (впровадження) не функціональних вимог

Реалізація (впровадження) не функціональних вимог діяльність складається з реалізації вимог, які випливають з атрибутів якості в частинах функціональності, надійності, зручності використання, ефективності та портативності (мобільності), і це описано в розділі про цілі архітектури. Реалізація цих вимог не обов'язково вимагає архітектора, але вимагає експерта з домену який відноситься до певного нефункціонального домену (наприклад, безпеки). У не гнучких налаштуваннях проекту відповідальність архітектора полягає в тому, що він зобов'язаний переконатися, що всі ці вимоги враховуються, але також можна накласти цю відповідальність і на всю команду.

Ці не функціональні вимоги реалізуються як функціональні вимоги. До того ж, необхідне вирівнювання з архітектурної діяльності, структурування, проекту для розуміння та проекту для змін. Решта реалізації вимагає експерта з домену щодо врахування вимог, а не архітектора. В висновку, вище згадана діяльність не потребує більш детального аналізу. Достатнім буде просто проаналізувати його архітектурні частини.

2.6.4 Проект для зрозумілості

Проект для зрозумілості тісно пов'язаний з структуруванням домену рішення. Структура є необхідною для відстеження величезної кількості інформації, проте існують і обмеження в структурі. Якщо зробити структуру більшою, ніж потрібно, то це також заплутує, оскільки в цьому випадку людський мозок не здатен відстежувати деталі структури. Отже, є важливим зберегти архітектурну структуру якомога простішою, щоб задовольнити мету зрозумілості.

Для досягнення цієї мети необхідно об'єднати дві концепції:

– початкова архітектурна структура повинна бути якомога меншою. тільки структура, яка необхідна для початку реалізації, повинна бути попередньо розроблена;

– архітектурна структура повинна бути спрощена, коли це можливо, протягом всього ЖЦ системи. це вимагає зусиль, щоб зберегти структуру системи простою. без явної протидії структура буде ускладнюватися з кожною зміною системи.

Джим Коплен з Гертрудом Бйорнвігом пропонують підхід до вирішення питання, що належить до початкової архітектурної структури.

Науковці радять відокремлювати «форму» і «структуру» системи:

– *«Форма»* – це сутність системи, розумова модель, яка стоїть за нею, інваріанти доменів-учасників.

– *«Структура»* – це суб’єкт, який може змінюватися протягом усього ЖЦ системи; отже, він не додає цінності для того, щоб включати його в початковий архітектурний проект.

Явне моделювання сутності системи допомагає створити початкову структуру, яка має досить стабільні інтерфейси. Через те, що первісна структура відображає основні концепції залучених доменів (предметних областей), система стає легшою для розуміння. Стабільність інтерфейсів також покращує змінюваність, оскільки зміни зберігаються локально без компрометації інтерфейсів.

Відокремлення такої початкової структури є явною роботою. Вона не появляється з повторюваного циклу реалізації (впровадження) та рефакторингу. Також, пошук адекватних абстракцій вимагає спеціального набору навичок та багато досвіду, тому що навіть якщо добре зрозуміло, що означає розмежування проблем, досі не досліджено, як найкраще проектувати і виконувати розподілення. Кілька евристик – це розподіл за відповідальністю та за різною швидкістю змін, організаційною структурою або розподіл за навичками – відомі (див. [12] для колекції), проте саме створення початкової структури є складним завданням, яке вимагає багато досвіду.

Загалом, можна використовувати цикл виникаючої архітектури, для того, щоб дозволити протікати еволюції сутності домену, і як правило, це вимагає більше часу і ресурсів, аніж явний підхід. Зазвичай, це вимагає декількох спроб і переробки проекту до того, як ця виникаюча розроблена архітектура сходиться до сутності домену, до того ж існує ризик того, що сутність домену ніколи не буде досягну[57] Це призводить до порушення мети архітектурної роботи, а саме звести до мінімуму загальні витрати і максимізувати загальне задоволення.

Другою концепцією, яка є необхідною для розуміння архітектурної структури, є постійне спрощення архітектурної структури. Це основна мета рефакторингу, який є частиною виникаючого циклу архітектури. Перш за воно впливає на «структуру» системи, а не на «форму». «Форма» містить в собі інваріанти системи і через те, зазвичай, не повинна сильно змінюватися після початкового проектування. "Структура", з іншої сторони, підлягає частим змінам і тому повинна бути розроблена з використанням виникаючого підходу до архітектури, попри те, щоб робити це явно.

Враховуючи досвід автора, цей підхід все ще несе певний ризик відсутності координації. Команди розробників, які є новими для виникаючої архітектури, часто не розподіляють свої проектні рішення одним з одним. Через це, одна і та ж вимога вирішується кількома різними способами, що зазвичай вимагає великих зусиль для уніфікації рішень пізніше для підвищення зрозумілості.

Загальним методом вирішення цього ризику є спілкування та обговорення проектних рішень у команді розробників для обміну знаннями. Альтернативно, можуть використовуватися архітектурні обмеження. Щоб уникнути подальшої роботи з уніфікації, визначаються та обговорюються необхідні архітектурні обмеження, і вся команда дотримується їх. Архітектурні обмеження допомагають уникати дублювання рішень без специфікації та деталізації архітектури, таким чином залишаючи достатньо місця для еволюції рішення.

2.6.5 Проект для змін

Проект для змін стосується майбутнього запиту на зміни. Формально система повинна бути відкритою лише для запитів на зміни, які впливають на систему протягом всього ЖЦ, щоб забезпечити ідеальний проект для змін.

Очевидна проблема полягає в тому, що майбутні запити на зміни не можуть бути передбачені.

Це часто призводить до наступних типів реакції:

– параліч аналізу, коли архітектори не наважуються приймати рішення, бо бояться пропустити проектне рішення, яке може стати важливим у майбутньому;

– надлишкові універсальні та гнучкі архітектури, коли рішення затримуються, зберігаючи структуру неспецифікованою та відкритою для якомога більшої кількості запитів на зміни в майбутньому;

– незнання, коли потенційні запити на зміни в майбутньому ігноруються. Замість цього система розроблена лише для відображення відомих вимог.

Всі реакції неоптимальні і порушують мету архітектури. Перший запобігає реалізації, оскільки для ефективного початку кодування необхідно прийняти рішення. Стандартною реакцією є те, що розробники починають кодувати без будь-якої базової архітектурної структури, яка також є недостатньо оптимальною.

Друга реакція компрометує розуміння. Кожна точка гнучкості в системі збільшує її складність, що ускладнює розуміння, але зрозумілість є передумовою змінюваності. Неможливо змінити систему належним чином, не розуміючи її. Таким

чином, занадто велика гнучкість компрометує розуміння і, у свою чергу, змінюваність. Важливою складовою архітектурної роботи є пошук розумного балансу між зрозумілістю та гнучкістю.

Третя реакція, порушуючи мету архітектурної роботи, ігнорує інформацію, яка зазвичай доступна під час розробки та може допомогти прийняти кращі рішення. Незважаючи на те, що неможливо точно передбачити майбутні запити на зміни, можна визначити ймовірності для певних типів запитів на зміни. Вони можуть бути використані для прийняття кращих рішень про те, де поставити точку гнучкості.

Для визначення ймовірних майбутніх запитів на зміни, можна використовувати такі інструменти:

– бізнес-сферу необхідно добре розуміти, адже більшість майбутніх запитів на зміни беруть початок у бізнес-домені. розуміння концепції цієї галузі полегшує створення змінних конструкцій;

– вивчити бізнес-стратегію, тому що,отримавши розуміння того, куди йде компанія і які їх головні цілі, можна очікувати велику кількість майбутніх запитів на зміни;

– потреба та драйвери зацікавлених сторін повинні розумітися якомога краще, адже вони створюють прагнення до змін, що призводить до запитів на зміни;

– треба проаналізувати тенденції розвитку бізнесу та IT (включаючи аналіз конкурентів). Розуміння того, де ринок збирається перейти на допомогу в розумінні майбутнього зовнішнього тиску на компанію, зазвичай призводить до запитів на зміни;

– формати семінарів на основі сценаріїв, наприклад, метод аналізу архітектурних компромісів (ATAM), або метод аналізу витрат та вигоди (CBAM), можуть бути використані для виявлення та визначення пріоритетів потенційних запитів на зміни дуже ефективним чином.

Використання цих інструментів допоможе виділяти напрямок змін – найбільш імовірні типи майбутніх запитів на зміни. Для найбільш ймовірних запитів на зміну повинні бути передбачені точки гнучкості. Це не завжди означає передбачити і забезпечити щось сконфігуроване. Часто достатньо розподілити обов’язки,

запровадивши відповідний інтерфейс, щоб зберегти майбутні зміни локалізовано. Оскільки підвищена гнучкість негативно вливає на розуміння, важливо не вводити занадто багато точко гнучкості.

Такий підхід не допомагає передбачати майбутні запити на зміни в усьому ЖЦ системи. Залежно від бізнес та IT-середовища системи, такий підхід зазвичай допомагає прогнозувати майбутні запити на зміну протягом декількох місяців абр до року, іноді трохи довше, ніж рік. Тим не менш, архітектурні рішення прийняті на основі цього поточного часу, тому що драйвери зміни запитів змінюються з часом, і в кінцевому підсумку деякі рішення стануть недійсними.

Тому архітектура та архітектурні рішення повинні регулярно переоцінюватися. Це означає, що новий напрямок змін необхідно виділяти, щоб з’ясувати, які точки гнучкості могли бути застарілими і які нові є потрібними. Використовуючи цей підхід, архітектура може бути збереженою і зміненою протягом усього ЖЦ системи.

Виділення напряму змін – це явна робота. Вона не з’являється з повторюваного циклу реалізації (впровадження) та рефакторингу. Можна також використовувати цикл виникаючої архітектури для адаптації архітектури до змінюваних вимог, але зазвичай це займе набагато більше часу і ресурсів, ніж явний підхід. Це, у свою чергу, порушує мету архітектурної роботи: звести до мінімуму загальні витрати і максимізувати загальне задоволення.

2.7 Явна та виникаюча архітектура

2.7.1 Порівняння явних і виникаючих архітектурних робіт

Необхідним тепер є протиставити сильні та слабкі сторони явних і виникаючих архітектурних робіт один одному та в результаті буде сформовано рекомендований спільний підхід.

В попередньому розділі вказано такі результати для питання, чи можна замінити архітектурну діяльність виникаючою архітектурою:

– Вирівнювання не може бути заміненим виникаючою архітектурою.

– Структурування повинно підтримувати цілі архітектурної роботи: реалізація (впровадження) нефункціональних вимог, проектування для розуміння та проектування для змін. Тож виникаюча архітектура діє проти цілі, а не структурування. Якщо структура документована для спілкування із сторонами, що необхідні і які не є розробниками, тоді ця документація не повинна бути кодом, тому це означає, що вона не є результатом виникаючого циклу реалізації (впровадження) та рефакторингу.

– Реалізація не функціональних вимог вимагає експерта для певного домену, а не архітектора. Тому ця діяльність не розглядається як основна архітектурна діяльність. Робота може бути зроблена архітектором, але не вимагає його.

– Проектування для розуміння може бути виконане виключно негайно, але це, зазвичай, порушує мету архітектурної роботи (максимізація загального задоволення зацікавлених сторін, мінімізація загальних витрат протягом всього життєвого циклу системи). Найкращий підхід полягає в тому, щоб розробити суть (ядро) системи явно і дозволити решті розвиватися, використовуючи виникаючу архітектуру та потенційно керуючись деякими архітектурними обмеженнями, щоб уникнути дублювання рішень розробки.

– Проектуванням для змін можна знехтувати, застосовуючи виключно виникаючий підхід, але це може порушити мету архітектурної роботи. Найращий підхід полягає в тому, щоб час від часу виділяти напрямок зміни (залежно від швидкості зміни системного середовища) для оптимізації проектних рішень.

Це в сумі призводить до порівняння явних і виникаючих архітектурних робіт, показаних на рис. 2.6. Враховуючи саме порівняння, на малюнку також зображено відносні зусилля різних архітектурних дій у порівнянні один з одним (на основі досвіду автора цього розділу):

– Вирівнювання є відносно великим зусиллям, особливо на ранніх етапах впровадження системи. Важливим є зібрати велику кількість відсутньої інформації, роз’яснювати неясні вимоги, щоб зробити їх реалізованими, вирішити суперечливі

та багато іншого. У певний момент часу потрібно виконати лише обсяг роботи по вирівнюванню, необхідний для безпечного запуску наступної ітерації. Це можна порівняти з роботою, яку має зробити власник продукту Scrum, щоб переконатися, що беклог продукту готовий до наступного спринту.

– Для створення проекту для змінюваності потрібно докласти відносно небагато зусилля. Більша частина необхідної інформації може бути зібрана під час роботи над вирівнюванням, а решта відсутня інформація зазвичай може бути зібрана досить швидко. Визначення напряму змін і прийняття рішення про додаткові точки гнучкості є простою діяльністю, якщо потрібна інформація є доступною.

– Проектування сутності системи («форми») загалом виконується відносно невеликими зусиллями тому , що більшість необхідних знань зібрані під час вирівнювання. Завдання полягає в тому, щоб знайти адекватні абстракції. Для цього необхідний певний набір навичок і багато досвіду.

– Створення деталізованої структури («структури») вимагає залучення більших зусиль, ніж для інших діяльностей.

Архітектурна діяльність	Вирівнювання	Проектування для змін	Проектування для зрозумілості	
			Сутність системи (Форма)	**Детально структурований (Структура)**
Відносні зусилля	Великий	Малий	Малий	Величезний
Явний підхід	(++)	(+)	(+)	(—)
Виникаючий підхід	(—)	(–)	(o)	(++)

Рис. 2.6. Порівняння явних і виникаючих архітектурних робіт

Досліджуючи явну і виникаючу архітектурну роботу, можливо зробити висновок, що ці два підходи доповнюють один одного. Попри те, що явна

архітектурна робота може охоплювати всі види діяльності, вона забезпечує найвищу цінність у вирівнюванні та проектуванні змін. Це також підходить для виділення сутності системи з не зовсім зрозумілим обмеженням як найкраще створити поділ. Ведення детального проекту в цілому є марною тратою часу і не має великого значення. Архітектор не потрібен для виконання роботи, яку часто можна передати розробникам, адже вона не дає йому достатньо часу для виконання іншої діяльності вищого значення. Зазвичай, декілька архітектурних обмежень є достатньо явною роботою в цій області.

З іншої сторони, виникаюча архітектура є достатньою частиною для створення зрозумілої та деталізованої структури. Обсяг робіт може бути розподілений точно по всьому проекту системи. Декілька архітектурних обмежень зможуть допомогти уникнути непотрібних дублюючих рішень. Сутність домену також може бути виділена з часом, використовуючи виникаючу архітектуру, але зазвичай для цього потрібен додатковий час та зусилля, при цьому, порушується головна мета архітектурної роботи. Виникаюча архітектура не надає підтримки для розроблення змін або вирівнювання (тобто існує високий ризик того, що мета архітектурної роботи порушується незадоволеними зацікавленими сторонами або збільшенням загальних витрат, якщо виникаючий підхід не підтримується додатковими заходами для забезпечення вирівнювання та розробки змін).

Виникає питання, розглядаючи цей контексті: «Чому власник продукту в Scrum, або користувач в разі XP (екстремального програмування) не повинен займатися вирівнюванням і виділенням напрямків змін?». Здавалося б, що архітектор і команда розробників могли б використовувати чистий виникаючий підхід.

Загалом, немає нічого поганого в тому, що власник продукту піклується про деякі аспекти архітектурної роботи, але враховуючи досвід автора цієї глави, є деякі суперечності, пов'язані з таким підходом:

– для великої кількості власників продукту важливим є лише функціональні вимоги. Вони міркують, що не несуть відповідальність за беклоги продукту, який складається лише з записів, які мають бізнес-цінність (значення), бо

нефункціональні вимоги будь-якого роду не несуть ділової цінності. Навіть якщо це твердження неправильне (безпека, доступність або продуктивність остаточно мають ділову цінність), зазвичай власники продуктів не думають про архітектурні потреби;

– власники проектів особливо відповідальні за досягнення цілей відповідного проекту та приймають на себе деякі обов'язки традиційної ролі менеджера проекту. Якщо брати до уваги не тільки відповідальність за проект, але й за успіх систем, то вони можуть в подальшому страждати через суперечливі цілі. Тому необхідно відокремити відповідальність за проект від відповідальності за систему;

– архітектурна розробка вимагає знань з розробки ПЗ, яких власники продуктів не мають. Власники продуктів часто є співробітниками відділу бізнесу або іншого відділу, що не є ІТ. Якщо людина не володіє необхідними навичками для конкретного завдання, людина прагне нехтувати його і зосередитися на різних завданнях, які вона здатна виконувати краще;

– архітектурна робота повинна виходити з команди. Або вся команда, а 64 окремі кваліфіковані члени команди мають поділитися своїми знаннями в команді. Власники продуктів, які є керівниками проектів, часто не вважаються частиною команди. Часто також виникає перешкода проходу потоку знань.

Підсумовуючи, можемо сказати, що іноді існує можливість дозволити власнику продукту піклуватися про вирівнювання і виділення напрямку змін, але з через причини, які були описані вище, в більшості проектів, це не спрацює так, як потрібно.

2.7.2 Спільний підхід

Додаючи інформацію попереднього розділу, можна сформувати спільний підхід до гнучкої архітектурної роботи, використовуючи особливі переваги явного та виникаючого підходів. Це призводить до загального спільного підходу гнучкої архітектурної роботи, показаної на рис. 2.7.

Розглянемо такі архітектурні заходи необхідні для перетворення вимог та потреб зацікавлених сторін у рішення та робочий код, з точки зору процесу розробки:

– для узгодження рішення свого завдання необхідним є зробити вирівнювання. Необхідно також розуміти зацікавлені сторони, роз'яснювати вимоги та вирішувати конфлікти; це також означає вивчення «мови» зацікавлених сторін і вимагає багато навичок з розробки ПЗ;

– проектування для змін складається з виділення напрямку зміни раз за разом (необхідна частота залежить від швидкості зміни задачі та проблемної області рішення); проект сутності домену також підтримує змінюваність пов'язаної з ним системи;

– для розуміння проектування потрібно частково зпроектувати сутність домену; виокремлення довгострокових стабільних частин системи зазвичай робить його набагато зрозумілішим, але найбільшою частиною роботи є створення детальної структури, вони, в свою чергу, необхідні для зберігання рішення зрозумілим;

– потрібно мати на увазі, що структурування ділиться на три частини: довгострокові стабільні частини, середньострокові стабільні частини та нестабільні частини. В свою чергу, довгострокові стабільні частини є сутністю («формою 65 системи, які потребують явного виділення.

Рис. 2.7. Спільний підхід до гнучкої архітектурної розробки з використанням виникаючої архітектури

Для визначення середньострокових стабільних частин та належних додаткових точок гнучкості, необхідно використовувати проект для змін, який також є явною діяльністю. Нестабільні частини розвиваються з використанням виникаючої архітектури, керуючись деякими архітектурними обмеженнями для уникнення дублювання рішень. Як було зазначено вище, нестабільна частина є значною частиною структуризації. Використовуючи підходи для цієї частини, робота поширюється краще, і потенційно дефіцитні навички, які необхідні для інших архітектурних заходів, не витрачаються даремно.

З точки зору архітектурного стилю роботи можна використати наступне:

1) Вирівнювання (регулювання), виділення напряму змін і проектування сутності домену зазвичай є виразною архітектурною діяльністю. Вони не виходять з «виникаючого» циклу кодування і рефакторингу. Незважаючи на те, що можна дозволити сутності домену вийти з «виникаючого» циклу, його явне проектування зазвичай відповідає цілям архітектурної роботи.

2) Створення детальної структури, а іноді й проектування сутності домену (з описаними раніше обмеженнями), може бути зроблено «виникаючим» чином,

залишаючи її «виникаючому» циклу реалізації та рефакторингу, що супроводжується деякими керівними принципами та деякими додатковими архітектурними обмеженнями, якщо потрібно.

З точки зору командної роботи можна зробити наступні спостереження:

1) Частини, що виникають, повинні бути виконані всією командою розробників.

2) Явні частини вимагають спеціального набору навичок та багатого досвіду. Зазвичай, лише декілька членів команди можуть піклуватися про ці завдання. Для кращого обміну знаннями ці члени команди також повинні виконувати звичайну розробку програмного забезпечення. Ґрунтуючись на досвіді автора, обмін знаннями найкраще працює, якщо люди працюють разом над завданням. Також важливо, щоб ці спеціально кваліфіковані люди не втратили зв'язок з рештою команди розробників.

Такий підхід різко знижує кількість явних архітектурних робіт, використовуючи потужність виникаючої архітектури. Таким чином, максимізується цінність загальної архітектурної роботи і найкраще використовуються навички залучених осіб.

ВИСНОВКИ ДО РОЗДІЛУ 2

У цьому розділі розглянуто твердження, що виникаюча архітектура може замінити явну архітектурну роботу. В якості передумови для аналізу визначено мету, діяльність та завдання архітектурної роботи. Після ретельної перевірки, вирівнювання, проектування для змін, та проектування для зрозумілості (яке складається з виділення сутності системи та деталізованого проектування) були визначені як архітектурні заходи, які необхідно проаналізувати щодо питання про те, чи може виникаюча архітектура замінити явну архітектурну роботу. Аналіз діяльності показав, що вирівнювання (регулювання) та проектування для змін не охоплюються циклами архітектури, які складаються з кодування та рефакторингу. Нехтування цією діяльністю порушує мету архітектури (тобто загальне задоволення зацікавлених сторін не буде максимальним, а загальні витрати не будуть мінімізовані протягом всього життєвого циклу відповідної системи). Виділення сутності («форми») системи також не охоплюється виникаючою архітектурою. Можна дозволити сутності системи використовувати виникаючу архітектуру, але це також може принести в жертву ціль архітектури, витрачаючи додатковий час і зусилля. Створення зрозумілого, деталізованого проекту дуже добре охоплюється архітектурою. Оскільки це, безумовно, найбільша діяльність з архітектурних робіт, має сенс використовувати для неї виникаючу архітектуру. Такий підхід краще поширює більшість архітектурних робіт і гарантує наявність потенційно дефіцитних навичок для інших архітектурних заходів, які не охоплюються архітектурою.

Спільний підхід до гнучкої архітектурної роботи був отриманий на підставі попередніх висновків. Він використовує архітектуру, що розроблюється, для деталізованого проектування, в той час як інші дії виконуються явно. Цей підхід добре розподіляє архітектурну роботу по всій команді, і потенційно дефіцитні люди, які мають необхідні навички та досвід, можуть зосередитися на заходах, які не охоплюються виникаючими циклами архітектури. Таким чином, agileвартості найкраще сприймаються шляхом максимізації значення, створеного за допомогою наявних навичок та досвіду.

РОЗДІЛ 3

РЕФАКТОРИНГ, РЕІНЖЕНІРИНГ ТА ПЕРЕПИСУВАННЯ ПРОГРАМ В ГНУЧКИХ ТЕХНОЛОГІЯХ РОЗРОБКИ

З кожним роком у сфері програмування відбувається все більше змін, які в результаті спричиняють до труднощів, які зустрічаються в архітектурі та які усунути стає складніше та дорожче по витратам. Прикладами таких змін можуть бути: зміна технологій розробки та інфраструктури, створення/введення/перегляд додаткових вимог, які і призводять до виникнення помилок(багів) і прийнятих помилкових рішень.

Але зосереджуватися важливо не на змінах, які відбуваються в системі, а саме на архітектурі ПЗ, так як якщо вона буде спрямована на внесення нових функцій, то вони можуть настільки збільшити і розширити даний проект, що в кінці кінців він стане некерованим і супроводжувати такий програмний продукт стане неможливим. Це стосується і певних випадків розробки архітектури, коли кожне помилково прийняте рішення призводить до незворотного результату та наслідків.

Аби уникнути такої долі проекту, обов’язковим фактором в архітектурі ПЗ є періодичне чергування проектних робіт з ітеративною оцінкою архітектури та проведення рефакторингу – метод покращення внутрішньої структури ПЗ без зміни зовнішньої поведінки системи. Якщо впровадити систематичне проведення рефакторингу у свій проект, це дозволяє розробникам ПЗ використовувати рішення, які є вже перевіреними та які приносять успішний результат. Тобто проведення оцінювання архітектури та рефакторингу допомагає уникнути ерозії проекту.

3.1 Види рефакторинга

3.1.1 Рефакторинг кода

Британський інженер-програміст Мартін Фаулер визначив термін рефакторингу як процес зміни вихідного коду ПЗ бfр зміни зовнішньої поведінки системи [16]. Наприклад існує метод, який дозволяє не дублювати код у різних методах, а окремо винести фрагмент коду, який повторюється, що значно полегшує супроводжуваність системи та її модифікацію. На рис.3.1 показано, що не змінюючи зовнішню поведінку, можливо за допомогою рефакторингу коду зменшити його складність.

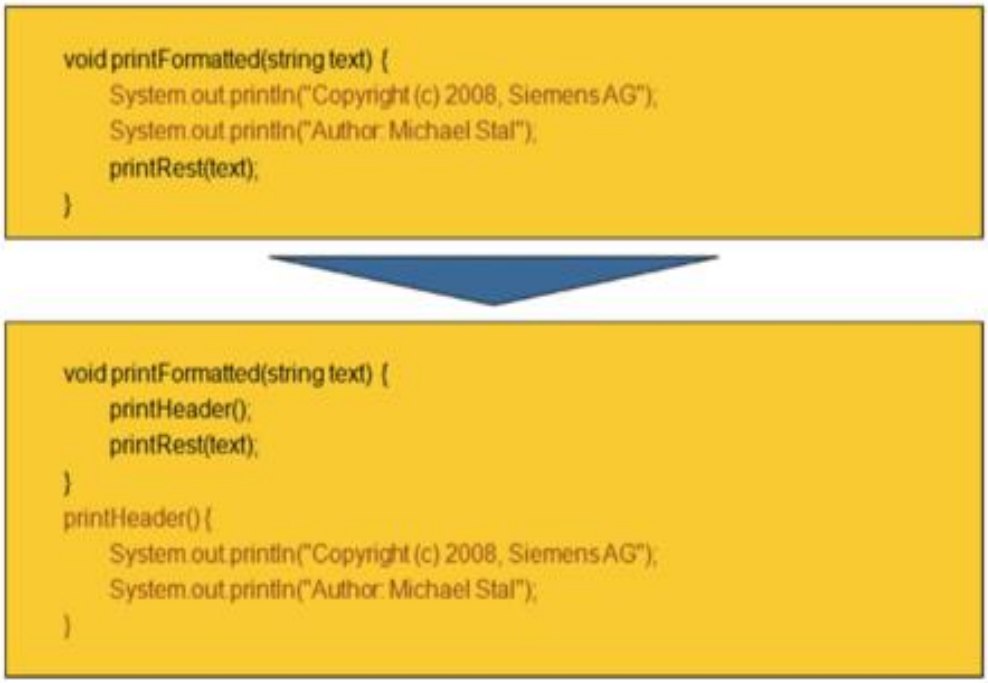

Рис 3.1. Метод винесення фрагменту коду у власні методи

Коли необхідно проводити рефакторинг? Кожного разу коли додається новий функціонал або виправлення помилки(багу). Але для того аби зрозуміти інженерам, яке саме покращення необхідне – були створені ознаки(їх ще називають «код з душком»), які допомагають визначити необхадність проведення рефакторингу коду:

– дублювання коду;

– методи класів, які займають велику кількість рядків;

– часте використання оператора switch.

У випадках, коли розробники зустрічають ознаки «коди з душком», вони мають виправити існуючі проблеми та поліпшити існуюче рішення.

3.1.2 Рефакторинг з патернами

Автор книги «Рефакторинг з використанням патернів» Джошуа Керівскій запропонував ідею рефакторингу з патернами, тим самим вніс стрімкий розвиток у сфері рефакторингу [16]. Основна мета його ідеї полягає у тому, аби використовувати шаблони (патерни) замість власних рішень, пояснюючи це тим, що якщо існує шаблон для певної проблеми, то він пропонує найкраще рішення аніж будь-який власний проект.

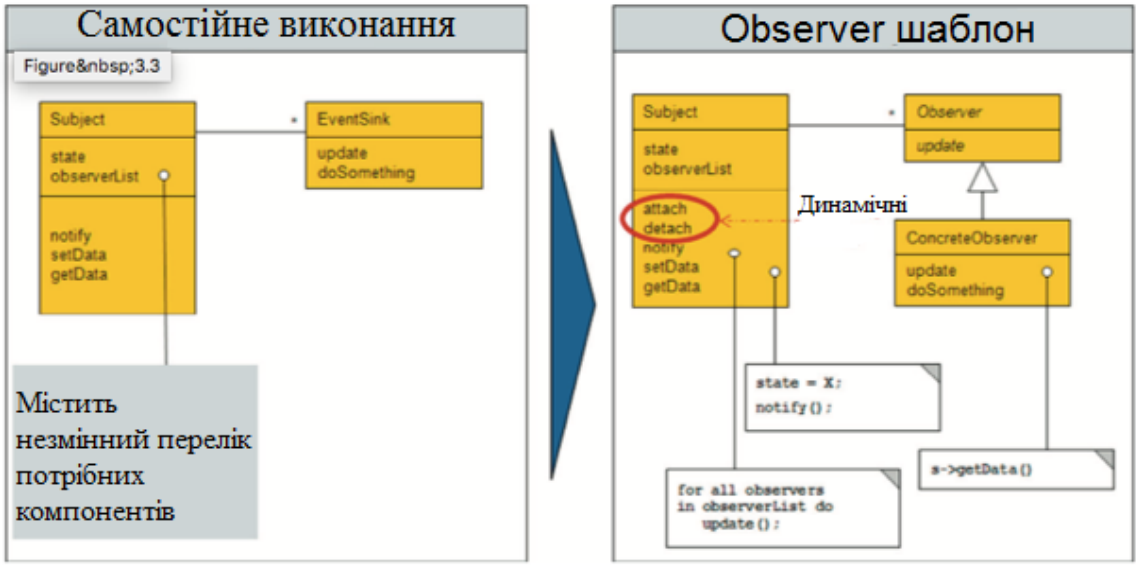

Рис. 3.2. Порівняння шаблону Observer та шаблону влансого рішення для рішення пробелми

Наприклад, можна не фізично з'єднувати посилання з підпискою на події, а використати патерн The Observer [16], який починає динамічне та, що важливо, гнучке з'єднання з важливими подіями (див. рис. 3.2). Рефакторинг з патернами в

цілому концентрує увагу не на кодуванні, а на проектуванні. Архітектура може залишитися такою як вона є або змінитися, але чи могли б ми використовуючи цей метод забезпечити рефакторинг іншим шарам абстракцій та іншим дисциплінам?

Важливо розуміти, що процес проведення рефакторингу може застосовуватися в обох випадках, зважаючи на його природу, без зміни семантики. У прикладі, що наведений на рис. 3.2, розробники, наприклад, могли використовувати власне рішення, підставою чого могла бути причина підтримки вбудованих пристроїв, які не мають динамічно змінюватися. Тобто всі рішення рефакторингу мають прийматися на основі бажаних вимог і властивостей якості, але ніяк не заради лише архітектури ПЗ або заради самого себе.

3.2 Причини рефакторинга програмного забезпечення

Мета рефакторингу покращити область, яка в цілому може поліпшити архітектуру. Для того аби визначити проблеми цієї області, а отже, і архітектури, і в подальшому провести рефакторинг було введено поняття «код з душком». Перелічемо деякі з ознак, які відносяться до такого коду:

1) *Дублювання артефактів проекту*: один з основних принципів рефакторингу – DRY, що означає не повторюватися. Якщо різним компонентам архітектури призначаються одні і ті ж обов'язки, то цей пинцип порушується. Загальні методи мають бути модульними, тому іншим питанням є вирішення того, який обсяг повторень є прийнятним для коду і відповідно необхідним, а яке повторення необхідно розглядати як проблемне місце, яке шкодить архітектурі проекту. У цьому випадку все залежить від проблеми та суміжних з нею результатів, яких можна досягнути використувуючи/не використовуючи принцип DRY.

2) *Неозначені ролі організацією сутностями*: в проекті за допомогою власних рішень командою розрбників було представлено декілька нових ролей – організатор та менеджер конференції. Керівники проектів відразу поставили питання про відмінність між цими поняттями та їх ролями. Тобто назви сутностей мають пояснювати їх роль та функцію в проекті, аби інші могли зрозуміти, за що

вони відповідають. Також кожному компоненту має бути призначений окремий обов'язок та окрема(одна) відповідальність, а не розподілені обов'язки між декількома сутностями. Тобто компонент має робити щось одне і виконувати це краще за всіх. Інакше можуть порушитися принцип рефакторингу з поділом проблем.

3) *Невиразна або складна архітектура*: надмірна складність в результаті може призвести до непотрібних та незрозумілих абстракцій, які в свою чергу можуть програмну систему зробити складною та невиразною. Це, наприклад, можуть бути об'єкти архітектури, які можуть бути незрозумілими; занадто деталізовані компоненти або навпаки занадто прості. Все це може ускладними розуміння архітектури ПЗ, яка має навпаки бути достатньо простою без перебільшення в ту чи іншу сторону.

4) *Все централізовано*: необіхдно з обережністю відноситися до централізованих підходів, навіть у випадках, коли такі підходи можуть здаватися доречними. Наприклад, при використання шаблону медіатора, шаблони зв'язку і залежності знаходяться в одному компонентів, що в результаті призводить до відмови або медіатора або концентратора, і на останок зменшує масштабованість. Тобто децентралізований підхід є більш підходящий, коли проблема по суті децентралізована.

5) *Самостійні рішення замість встановлених кращих практик*: зазвичай програмісти використовують свої власні рішення, а не готові перевірені кращі рішення. Але готові рішення часто перевершують самостійні. Як наприклад на рис.3.3 наведений шаблон Observer, який пропонує гарне рішення для проблем повідомлення про події. Але бувають випадки, коли власне рішення є кращим. Прикладом можуть слугувати системи в умовах реального часу, де динаміка непотрібна. Таким чином програмісти створюють власне рішення.

6) *Занадто універсальний проект*: деякі патерни(наприклад, патерни стратегії) дозволяють відтянути процес змінюваності на деякий період, але якщо ними зловживати – це матиме погані наслідки для виразності та переносимості системи. Наприклад, часте використання такими патернами, як стратегія,

спостерігач(Observer) та перехоплювач призведе до того, що буде важко формувати, розвивати та створювати архітектуру ПЗ. Тобто архітектура має бути універсальною, конфігурованою та характерною, але в міру необхідності, без перебільшень.

7) *Асиметрична структура або поведінка*: асиметрія вказує на проблемні місця в архітектурі. Існує два види симетрії: поведінкова симетрія і структурна симетрія. Поведінкова симетрія включає в себе такі дії як відкритий метод (вимагає методу close), транзакції яка починається (вимагає метод фіксації/відкату/вилки та об'єднання). Тобто іншими словами, якщо дужка відкривається, має бути дужка яка закривається. А структурна симетрія вимагає аби схожі проблеми вирішувалися схожими патернами. Наприклад, коли потрібно повідомлення про події, треба використовувати шаблон Observer. Звісно, бувають випадки коли асиметрія є необхідною, а симетрія не підходить для забезпечення якості архітектури: наприклад, коли розподіл і звільнення ресурсів має проводитися різними компонентами. Тому в теорії симетрія є показником гарної/поганої архітектури, але це не завжди так на практиці.

8) *Цикли залежностей*: цикли залежностей між компонентами архітектури зустрічатися не повинні, так як така ситуація свідчить про негативний вплив на тестування, модифікацію та виразність. Тому архітектор при тестуванні або модифікації компонентів в такому циклі, не може не проаналізувати інші компоненти такого циклу.

9) *Порушення політики проектування*: слід уникати такої ситуації, наприклад, використання вільної ієрархії, замість суворої. У першому випадку різні спеціалісти по-різному можуть вирішити проблему, що призведе до меншої видимості та виразності.

10) *Неправильний поділ функціоналу*: неправильний поділ функцій з підсистемою може стати причиною надмірної складності. Коли функції поділені вірно, а зв'язок між підсистемами є невеликим, це значно покращує єдність системи. Тому якщо існує забагато або замало підсистем, це може слугувати показником неправильного розбиття функціоналу на підсистеми.

11) *Непотрібні залежності*: непотрібні залежності призводять до надмірної складності і відповідно впливати на продуктивність та змінність системи. Тому коли метрики циклічності або складності значно збільшуються між двома ітераціями, це означає, що певні додаткові або непотрібні залежності мають бути виделані.

12) *Неявні залежності*: до надмірної складності також можуть призвести залежності, які недоступні в модулях архітектури. Коли розробники додадуть неявні залежності, при цьому не повідомивши про це нікому, інші розробники ПЗ не знаючи про ці залежності можуть порушити реалізацію та прозорість архітектури. Часте використання глобальних змінних в одному шаблоні є одним з яскравих прикладів такого «коду з душком». Іншим прикладом може слугувати порушення «вищих» рівнів ієрархії в архітектурі, які насправді мають досить строгий підхід до ієрархії. Якщо вищий рівень дає збої кожного разу, коли змінюється нижчий рівень збереження, це може бути причиної саме неявних залежностей.

Але «код з душком» це не доказ архітектурних проблем, вони вказують на те, що таку проблеми можуть існувати. Асиметрія, використання шаблону стратегії, дублювання коду – вони можуть знадобитися у тому чи іншому конкретному випадку, це залежить від контексту проблеми.

3.3 Тактика застосування рефакторингу

Розглянемо систему управління з прямокутними формами, що моделюють компоненти архітектури (рис. 3.3). Архітектори ПЗ в даному випадку під абстрактним сховищем мають на увазі всі види сховищ, де за допомогою певного алгоритму сратегії можна додати або вилучити елементи.

Архітектори представили абстракцію «транспортний шлях» для того аби доставити предмети між різними точками. Додавання такого шляху з додатковими компонентами і відносинами звісно призвело до створення складного ПЗ. Після детального аналізу стає зрозумілим, що шляхи нагадують абстрактні сховища зі зв'язаною стратегією, яка в свою чергу визначає вид транспорту. Тому усуваємо абстрагування транспортного шляху і розглядаємо транспортне обладнання як

особливий вид сховища. Завдяки наведеній стратегії архітектура даного сховища значно поліпшилася. З однієї сторони у конфігурації (наприклад, конфігурації бетонних сховищ), з іншої – змінюваність (наприклад, обмін конкретним сховищем або стратегією).

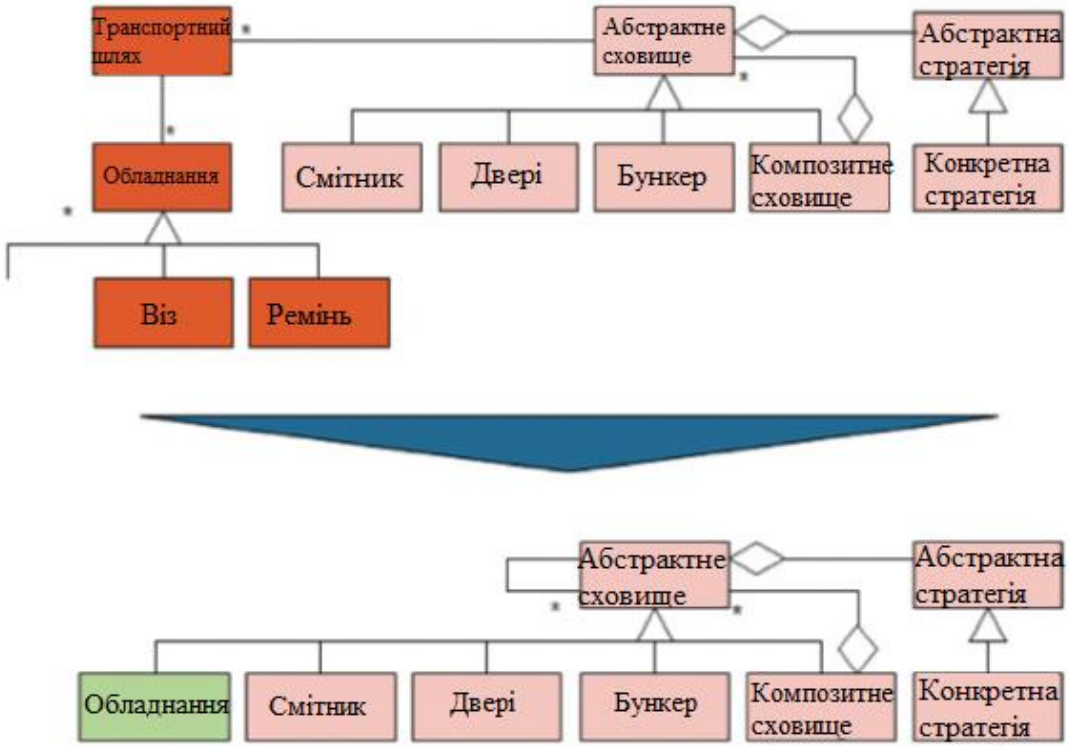

Рис. 3.3. Система управління з прямокутними формами, що моделюють компоненти архітектури

Якшо б рефактиринг не застосували у ситуації на рис. 3.3, то абстрації (транспортний шлях) були б уточнені та розширені пізніше, і в результаті збільшилося б поширення інших додаткових артефактів, введення нових залежностей в нступні ітерації. Тому необхідно якомога швидше вирішувати дану проблему шляхом рефакторингу, оскільки якщо переносити вирішення цього питання, то позбутися від неї буде складніше та дорожче.

В загальному цей підхід рефакторингу і є спільним рішенням для великої кількості проблем, які зустрічаються. Коли розробники ПЗ потрапляють у ситуацію, коли необхідно ввести нову абстракцію підсистемам, інтерфейсам або компонентам,

вони мають перевірити чи насправді вони необхідні або необхідно виконувати реструктуризацію проекту.

Коли проблема повторюється, рефакторинг вирішує цю проблему шляхом урівноваження різних пропозицій та обирає найкраще конкретне рішення. Тобто таке рішення є вже перевіреним практикою для певного кола проблем, яке розглядається як структурний шаблон перетворення (рис. 3.4) або, як його ще називають, патерн рефакторинга.

Важливо розглядати реорганізацію архітектури як додаток сруктур перетворення архітектури, який допомагає покращити якість ПЗ в певному контексті проблеми. Проблема задається з використанням атрибутів якості, сил та «коду з душком». Опис шаблонів допомагає організації рефакторинга в каталогах і зрозуміти який саме шаблон підходить для вирішення певного питання.

Лише деякі складові шаблону продемонстровані Патлета в прикладі на рис. 3.4. Але в повний опис шаблону необхідно додати такі розділи, як «Варіанти», «Реалізація» та «Наслідки» [4].

Назва

- видалити непотрібні абстракції в ієрархіях абстракції

Контекст

- видалення непотрібних абстракцій проекта після розширення системи

Проблема

- мінімалізм – важлива ціль архітектури ПЗ, оскільки саме мінімалізм збільшує простоту та виразність;
- якщо архітектура ПЗ включає абстракції, які можна розглядати як абстракції, отримані з іншої абстракції, тоді рекомендується видалити їх;

Загальна ідея вирішення

- визначте, чи існують абстракції або артефакти дизайну, які також можуть бути отримані з інших абстракцій;
- якщо це так, видаліть непотрібні абстракції та отримайте залежність від існуючих абстракцій.

Попередження

- Занадто не узагальнюйте (наприклад, введення одного рівня ієрархії: «Всі класи створюються безпосередньо з об’єкта»).

Рис. 3.4. Загальна тактика рефакторінга

3.4 Забезпечення якості при проведенні рефакторингу архітектури

Розуміючи, що рефакторинг можна застосовувати до коду, виникає питання, чи можна використати його до схем баз даних та UML діаграм, тобто до інших артефактів програмування. Через постійне зростання та еволюції SA архітектура ПЗ може бути областю для рефакторинга. Тому оцінка архітектури ПЗ [12, 13] і рефакторинг мають постійно чередуватися у всіх ітераціях (рис. 3.5).

Треба пам’ятати, що є 2 типи якості архітектури: внутрішня та зовнішня архітектура. Якість внутрішньої охоплює структурні аспекти (симетрія, згуртованість та зв’язок), інструменти оцінки архітектури та програмні метрики; якість зовнішньої – атрибути якості, що визначені в ISO / IEC 25010. Якщо казати про показники якості, то в загальному вони належать до внутрішньої якості, хоча можуть покращити і зовнішню (наприклад, продуктивність), коли непотрібні шари видаляються архітекторами побічно з їх архітектури. Внутрішнє поліпшення якості

(показники якості архітектури) вимагає проаналізувати плив їх на зовнішню якість(атрибути якості) [12]. Але основна мета рефакторингу – це покращити якість ПЗ.

Рис. 3.5. Оцінка та вдосконалення (рефакторинг) архітектури відбувається на кожній ітерації

Покращити якість внутрішньої архітектури можна за допомогою моделей та патернів рефакторингу. Далі перелічимо індикатори внутрішньої якості архітектури:

– *Економіка*: архітектура ПЗ має бути простою (проте не спрощенною) і містити саме ті артефакти, що потрібні для досягнення цілі розввитку.

– *Видимість*: архітектурні компоненти та залежності між ними мають бути легко зрозумілими та не мають бути прихованими.

– *Інтервал*: в архітектурних об’єктах мають ефетивно і продуктивно розподілятися обов'язки. Це можливо завдяки правильному поділу проблем.

– *Симетрія*: існує два види симетрії – поведінкова та структурна. Перша означає, що і для кожної «починаючої транзакції» повинен бути «відкат» або «фіксація», а для кожного «відкритого» оператора повинен бути «закритий». Друга означає, що в програмній системі архітектори забезпечували таке ж рішення. Тобто зазвичай на проблеми в дизайні вказує асиметрія.

– *Поява*: ціле більше, ніж сума його частин. Якщо в системі наявні прості частини, вони в цілосності можуть призвести до складної функціональності і це набагато простіше, аніж складні частини впроваджувати у складні артефакти.

Індикатори якості архітектури ПЗ, так само як і «код з душком» можуть вказувати на проблемні місця, але лише вказувати, так як вони не є гарантією поганої внутрішньої якості. Про це мають пам’ятати архітектори, коли такі індикатори виявляють.

Змінюваність, продуктивність та експлуатаційні характеристики також може покращити рефакторинг. Наприклад, якщо архітектура має непотрібні абстракції та залежності, непотрібні шари, то це у свою чергу зменшує змінюваність та продуктивність, а використання патернів рефакторинга допомагає визначити проблеми(непотрібні абстракції, шари), усунути їх та покращити змінюваність та продуктивність. Але необхідно розуміти де необхідно приміняти рефакторинг, а де ні.

3.5 Послідовні етапи процесу рефакторингу архітектури

Як вже згадувалося раніше, всі дії рефакторингу мають проходити ітеративно та систематично (див. рис. 3.5). Наведемо приблизну схему проведення рефакторингу:

1) *Оцінювання архітектури*: визначення «коду з душком» та проблем проектування архітектури. Це особливо актуально, наприклад, коли архітектура має задовольняти особисті атрибути якості. Необхідно перелічити всі ідентифіковані архітектурні проблеми та створити список (інструменти оцінки коду, управління його якістю, методи огляду архітектури та інше).

2) *Встановлення пріоритетів*: необхідно проставити пріорітети всім ідентифікованим архітектурним проблемам та вимогам, які впливають на них. Всі

проблемні питання стратегічного проектування мають бути вирішені перед тим як переходити до тактичних областей. А всі артефакти, які мають важливі вимоги, мають мати високий пріоритет.Таким чином проблемні місця в архітектурі розподіляються за пріоритетами та обсягами.

3) *Вибір*: для кожної з проблем, починаючи з тих, яким надані найвищі пріоритети, необхідно виконати наступні дії:

a. Обрати необхідний шаблон рефакторингу, тобто такий, що розглядає і внутрішню і зовнішню якість і відповідно до цих аспектів вирішує проблему.

b. В залежності від мети проектованої системи, якшо існує декілька шаблонів, необхідно обрати той, що дозволить досягнути цілей системи та позитивно вплинути на зовнішні якості.

c. Якшо жоден із наведених шаблонів не підходить для вирішення проблеми, треба повернутися до звичайного перепроектування архітектури.

4) *Забезпечення якості*: необхідно перевіряти для кожної програми, чи рефакторинг не змінює семантику системи. Існує 3 варіанти забезпечення якості:

a. Формальний підхід – дуже користний для критичних з безпеки систем. Можливо довести цим підходом, що структурне перетворення не змінило семантику.

b. Оцінка архітектури – необхідно провести огляд архітектури/проектування/огляд коду(якщо вже наявна реалізація) для перевірки якості.

c. Тестування – важливий аспект перевірки якості ПЗ у випадку якщо архітектура реалізована. Тут можна використовувати для визначення якості внутрішньої архітектури показники якості ПЗ.

Наявних інструментів рефакторингу, на сьогоднішній день, можливо, бракує, але їх достатньо для деяких етапів цього процесу. Наприклад, на етапі аналізу метрики та додатки допомагають визначити «код з душком».

Для гнучкої розробки процес рефакторингу є особливо важливим та корисним. Наприклад, шляхом включення рефакторинга діяльності спринту можна інтегрувати

реорганізацію архітектури. Архітектуру мають перевірити ще раз архітектори, а власники ПЗ та тестувальники – що система відповідає вимогам після рефакторингу. Після того як у конкретному спринті впроваджуються історії, розробникам ПЗ необхідно визначити «код з душком» та інші проблеми з якістю, тому проводиться оцінювання архітектури.

Реорганізація має проводитися 1 раз за ітерацію (якщо це буде відбуватися частіше, в архітектурі можуть відбуватися неконтрольовані та часті зміни, якщо рідше – вирішувати проблеми в архітектури стане складніше та довше в часових рамках), на відміну від рефакторинга коду, який проводиться у повсякденній роботі кожного дня.

До дати релізу рефакторинг питань некритичної архітектури не застосовується. Але якщо по цій причині не застосовувати певний конкретний рефакторинг, то його необхідно буде провести у наступній ітерації, так як він стає предметом проектної заборгованості. Наприклад, в Scrum методології, якщо певну проблему не розглянули в поточному спринті, то вона має бути збережена та підтримана в беклоге(заборгованість).

3.6 Проведення рефакторингу із застосуванням архітектурних патернів

3.6.1 Розривання циклу залежності

Централізовані монітори допомагають агентам в телекомунікаційній мережі управління визначити актуальний стан обладнання апаратного і ПЗ, тобто на кожному мережевому вузлі оператори управляють та контролюють їх стан. Коли відбуваються певні збої, агенти повідомляють про них, використовуючи передачу на основі подій. Але так як основною частиною цих повідомлень про події є відмітка часу, яким чином оператори, враховуючи синхронність годин в розподіленому середовищі, можуть прив’язати ці позначки до подій?

Архітектори, на жаль, використовують монітори для створення відмітки часу, вводячи цикл залежності (рис. 3.6). Цю проблему можна вирішити декількома способами: можна змінити одну із залежностей, шляхом введення механізму впровадження залежності або ще раз перепризначити обов'язки шляхом додавання додаткових компонентів архітектури (рис. 3.7).

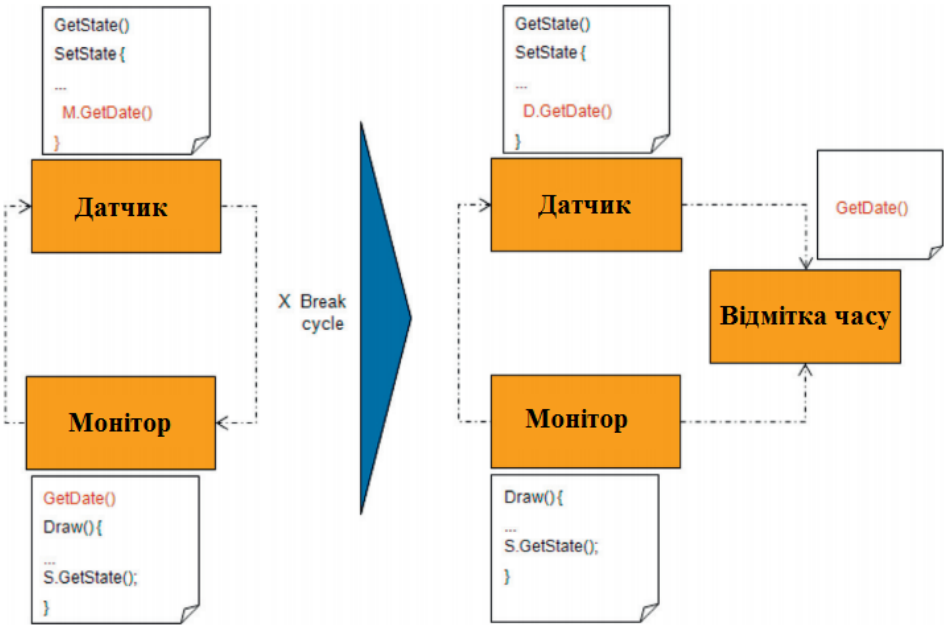

Рис. 3.6. Цикл залежності та трансформація циклу, що зменшує керованість, контрольованість і модифікованість

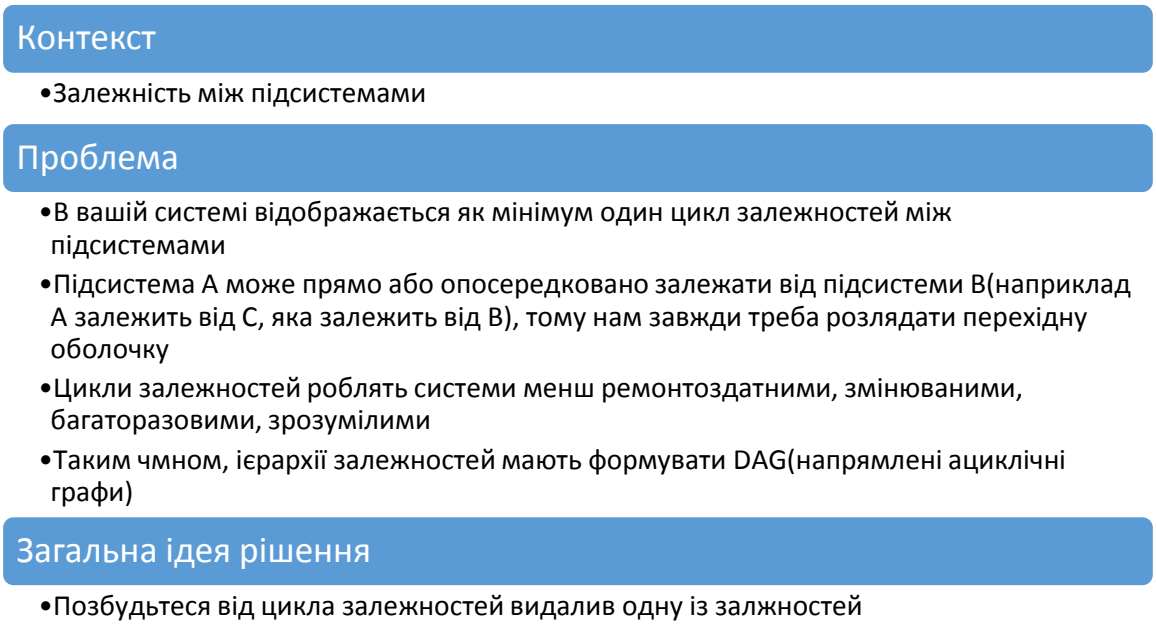

Рис. 3.7. Схема зразка рефакторінга архітектури для розриву циклу залежності

3.6.1 Розділення підсистем

До прикладів метрик архітектури можна віднести зчеплення та єдність. В підсистемах архітектури зчеплення між їх частинами може бути занадто тісним, що проводить в результаті до високої єдності, а зчеплення має бути навпаки вільним.

Майже всі компонент показали високу ступінь єдності у розвитку власної підсистеми інфраструктури контейнерів для об’єднаної системи зв'язку, з одним виключенням: засіб зв'язування комунікацій був слабо пов'язаний лише з іншою чатисною котейнера.

Тому було прийнято рішення архітекторами розділити підсистему (рис. 3.8):

1) підсистема для фаткичного контейнера;
2) підсистема для розподілу проміжного ПЗ.

Якщо відокремити комінікаційні обов'язки та контейнерні, то з часом в архітекторів буде можливість розширити/обміняти комунікаційну інфраструктуру без влпиву на контейнер.

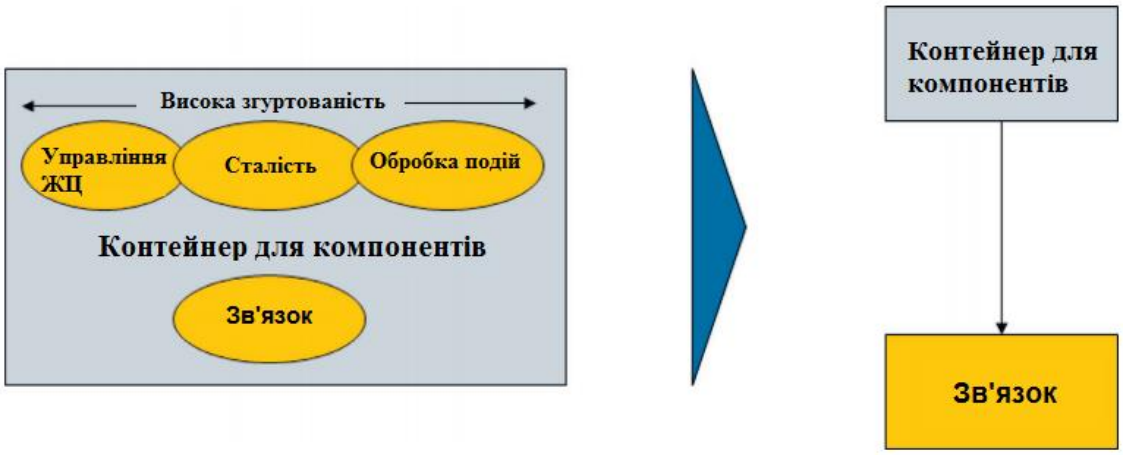

Рис. 3.8. Якщо компоненти в архітектурній підсистемі є слабо зв'язаними з іншими компонентами – це вказує на потенційне розбиття на безліч підсистем

В загальному в якості ідентифікації того коли є необхідність виконувати розділення або об'єднання підсистем можна використовувати ступінь зчеплення та єдності. Для розділення підсистем наведена схема патерна рефакторингу архітектури на рис. 3.9.

Контекст

- Згуртованість всередині підсистеми

Проблема

- Усередині підсистеми взаємозалежність (згуртованість) повинна бути високою;
- Між двома підсистемами в програмній архітектурі ступінь зчеплення повинен бути досить слабким. Якщо зчеплення між деякими частинами є вільним, то деякі проектні рішення можуть бути сумнівні, рекомендується змінити це щоб отримати кращу модульність і зрозумілість;
- Інша потенційна проблема це підсистеми / компоненти з дуже великою кількістю обов'язків

Загальна ідея рішення

- Вільне зчеплення в рамках підсистеми має на увазі, що функціональність може бути розділена на кілька підсистем;
- Таким чином, визначте області з високою зв'язністю в підсистемі. Всі ці області з низькою зв'язністю є кандидатами щоб стати самостійною підсистемою.

Рис. 3.9. Схема патерни рефакторінга архітектури для поділу підсистеми

3.7 Труднощі застосування рефакторинга архітектури

Можна зробити висновок, що проведення рефакторингу архітектури ПЗ є необхідним процесом при розробці ПЗ, але при його проведенні можуть виникати різні труднощі, про які архітекторам важливо знати наперед. Існує чотири області, де можуть з'явитися проблеми:

1) *Управління та організація*: у виробництві та управлінні проектами часто вважають, що нові можливості в інженерії ПЗ є найважливішими. Підхід, коли проект архітектури виконують лише з ціллю уникнення можливих помилок, не бере до уваги безперервну змінюваність. По-перше, архітектори ПЗ не знають детально всіх вимог, що дозволяє використовувати лише існуючі знаня. Коли знання поповнюються, то попередні рішення вже можуть бути не актуальними, відповідно вдосконаленими. По-друге, поступове зростання проекту постійно вимагає забезпечення якості всіх артефактів та рефакторингу архітектури.

Основною проблемою в даній ситуації є те, що лише після завершення проекту, за допомогою рефакторингу та тестування можна підтвердити його значення. Тому не треба очікувати від рефакторингу швидкої окупності інвестицій, треба пам’ятати, що ведення частих перевірок якості та внесення покращень є дешевшим варіантом, аніж нехтувати забезпеченням якості. Рішенням для даної проблеми є test-driven development (TDD) – розробка через тестування, так як саме цей підхід дозвояє зазделегідь знайти проблемні місця.

Іншою проблемою є невідповідність організації, яка часто керує архітектурою(відповідно до закну Конуея). Тому погана організація може призвести до поганої архітектури. При реорганіхації архітектури, якщо обов'язки в організаційних підрозділах розподілені неправильно, то працювати разом вони не зможуть (наприклад, коли підрозділи мають різні цілі).

2) *Процес розвитку*: підхід рефакторингу має бути повністю включений у загальний процес розвитку проекту, аби була можливість планувати необхідні ресурси для виконання цілей рефакторинга та чітко розуміти які обов'язки по рефакторингу які спеціалісти будуть виконувати (тестори та архітектора ПЗ).

Наприклад, існує менеджер по тестах, який має знати рефакторинг на такому рівні, аби він міг перевірити правильність його проведення та якість системи ПЗ.

3) *Технологія та інструменти*: через те що інструментів бракує для підтримки рефакторингу архітектури ПЗ, його необхідно проводити самостійно вручну, що збільшує відсоток допуску помилок та складність виконання. А пошук та вирішення проблем в архітектурі на більш пізніх етапах – ще більш підвищує кількість помилок та виконання.

Якщо доступний варіант використання каталогу або системи патернів, то архітектори можуть ефективно покращити архітектуру ПЗ, якщо ні – можна створити власну колекцію, хоча це і більш складний варіант і не завжди кращий.

4) *Застовування*: якби проблема розвивалася таким чином, що методи рефаткоринга могли вирішити ознаки, а не причини, то реінженіринг або переписування були б більш ефективними та доречними. Коли методи рефакторингу не покращують якість або коли поведінка системи має бути змінена, то такий сценарій завжди підходить до виконання і вказує, що рефакторинг не підходить у даному випадку.

Якщо узагальнити, архітектори ПЗ використовують рефакторинг і каталог патернів у більшості випадків [9]. Велика кількість цих проектів поширили патерни рефакторинга, а учасники проекту визнали за необхідне зробити доступним такий каталог. Тож використання рефакторинга архітектури може дійсно покразити якість архітектури та зменшити витрати компанії.

3.8 Рефакторинг, реінженіринг, перепрограмування

Покращити проект локально, на тактичному рівні, допомагає рефакторинг архітектури ПЗ. Але коли система ПЗ вже постраждала від сильної ерозії, то рефакторинг може вирішити лише ознаки, але не проблему її виникнення. Цю ситуацію розробники ПЗ можуть розпізнати – коли інтенсивний рефакторинг не призводить до очікуваного результату. У такій ситуації лише одного інструменту

рефакторингу не достатньо для відновлення архітектури, необхідно використовувати реінженіринг або переписування замість цього (табл. 3.1).

Таблиця 3.1

Порівняння рефакторинга, реінженіринга та переписування

	Рефакторинг	Реінжиніринг	Переписування
Процеси	• Багато місцевих ефектів • Перетворення структурии • Збереження поведінки/ семантики • Можлива зміна архітектурних рис	• Системні ефекти • Розбирання / повторна зборка	• Системні або локальні ефекти • Дорога заміна цілої системи з новою реалізацією
Результати	• Покращена структура • Однакова поведінка	• Нова система	• Нова система або компоненти
Покращення сторони	• Розвиваюча	• Функціональна • Оперативна • Розвиваюча	• Функціональна • Оперативна • Розвиваюча
Драйвера	• Складна еволюція проекта/коду • При виправленні помилок • Після проекта та «кода з душком»	• Рефакторинга недостатньо • Виправлені помилки через ряби • Новий функціонал та вимога • Змінений бізнес-кейс	• Рефакторинга та реінжиніринга недостатньо • Нестабільний код та проект • Нові фкнціональні експлуатаційні вимоги • Змінений бізнес-кейс
Коли	• Частина повсякденної роботи • В кінці кожної ітерації • Спеціалізований рефакторинг ітерацій у відповідь на відгуки • Це третій крок TDD	• Необхідний надійний проект	• Потрібні спеціальні зусилля або проект, в залежності від об’єму

Реінженіринг від рефакторингу відрізняється тим [9], що перший впливає на систему ПЗ системно, у першому етапі вся система перепроектована та її компоненти оцінені, використовуючи SWOT (Strengths, Weaknesses, Opportunities, andThreats – Переваги, Слабкі місця, Можливості і Загрози). Розробники ПЗ будуть адаптувати цінні компоненти, відповідно повторно використовуючи їх. На наступному етапі вони будуть вже частиною системи. Тут вступає, зазвичай, рефакторинг для адаптації компонентів.

У випадку, коли дії по реорганізації компонента або системи перевищують дії по відновленню, то єдиним варіантом є проведення переписування. Але використовувати його доречно лише тоді, коли рефакторингу та реінженірингу буде недостатньо або вони взагалі недоречні.

ВИСНОВКИ ДО РОЗДІЛУ 3

Розвиток і систематичне поширення системи ПЗ є необхідною, але лише однією стороною монети. Іншою стороною є проблеми, з якими стикаються архітектори ПЗ, так як вони відповідальні за уникання складності архітектури існуючого ПЗ. Для цього вони використовують та посилюють оцінювання архітектури та тестування.

Рефакторинг дозволяє покращення структури архітектури, не змінюючи поведінку системи. Використовується він, коли був виявлений «код з душком» при оцінюванні архітектури, мінімум один раз за ітерацію (мається на увазі у гнучкому або ітеративному процесі розробки). Тому проведення рефакторингу вкрай необхідне до та після додавання нового артефакту до проекту ПЗ або коли архітектори визначають важливі проблеми. Таким чином, можна виявити та усунити неправильні проектні рішення ще на початку, гарантуючи покращення якості в архітектурі ПЗ. Рефакторинг є обов'язковим в TDD.

Але несистематичне проведення рефакторингу може призвести до ще більших проблем – наприклад, коли використовують неправильний патерн рефакторинга, який не розглядає початкові цілі проведення рефакторинга, неохідні якісні ознаки (аспекти реального часу та інше). Тому рефакторинг перш за все вимагає його проведення систематично.

Рефакторинг архітектури ПЗ залишається важливим і цінним інструментом управління складністю програмних систем, так як їх розмір та складність зростають з кожним роком. І якщо використовувати процес рефакторингу систематично, то він може слугувати для архітекторів гарантією збереження ПЗ у хорошому стані.

РОЗДІЛ 4

ПРОЕКТУВАННЯ ПРОГРАМНОГО КОМПЛЕКСУ АНАЛІЗУ ЯКОСТІ ПРОГРАМНОЇ АРХІТЕКТУРИ ТА ПРОВЕДЕННЯ РЕФАКТОРИНГУ

4.1 Проектування процесу проведення рефакторингу та створення UML діаграм

Незалежно від методології розробки, яку ви застосовуєте, першим етапом розробки буде формулювання вимог до продукту. Набір вимог до продукту є технічне завдання(ТЗ), при цьому вимоги діляться на функціональні (бажана функціональність) і нефункціональні (вимоги до обладнання, операційної системи і т.п.). У мові UML для формалізації функціональних вимог застосовуються діаграми використання.

Діаграму варіантів використання є сенс будувати під час вивчення ТЗ, вона складається з графічної діаграми, яка описує дійові особи і прецеденти, а також специфікації, що представляє собою текстовий опис конкретних послідовностей дій (потоку подій), які виконує користувач при роботі з системою. Специфікація потім стане основою для тестування і документації, а на наступних етапах проектування вона доповнюється і оформляється у вигляді діаграми (в рамках ICONIX використовується діаграма послідовності, але в UML для цього є також діаграми діяльності). Крім того, use-case діаграма досить проста, щоб її міг зрозуміти замовник, отже ви можете використовувати її для узгодження ТЗ (адже діаграма описує функціональні вимоги до системи).

На діаграмі використання зображуються:

- актори – групи осіб або систем, що взаємодіють з нашою системою;
- варіанти використання (прецеденти) - сервіси, які наша система надає

акторам;

– коментарі;

– відносини між елементами діаграми.

Найбільш правильний порядок побудови діаграми наступний:

1) виділити групи дійових осіб (які працюють з системою по-різному, часто через різні прав доступу);

2) ідентифікувати якомога більше варіантів використання (процесів, які можуть виконувати користувачі). При цьому не слід ділити процеси занадто дрібно, потрібно вибирати лише ті, які дадуть користувачеві значущий результат. Наприклад, касир може «продати товар» (це буде прецедентом), проте «введення штрих-коду товару для отримання ціни» самостійним прецедентом не є;

3) доповнити прецеденти словесним описом (сценарієм):

a. для кожного прецеденту створити розділи: «головна послідовність» і «альтернативні послідовності»;

b. при складанні сценарію потрібно наполегливо ставити замовнику питання «що відбувається?», «що далі?», «що ще може відбуватися?» і записувати відповіді на них.

c. Варіант використання – це взаємодія між користувачем та проектованою системою, який характеризується такими властивостями:

– варіант охоплює деяку очевидну для користувачів функцію;

– варіант може бути як невеликим, так і досить великим;

– варіант вирішує певне конкретне завдання користувача.

Тому виділимо основні варіанти використання для аналізу необхідності та проведення рефакторингу:

– визначення необхідності проведення рефакторингу;

– аналіз характеристик «коду з душком»;

– визначення варіанта рефакторинга архітектури (патерни чи власні рішення);

– визначення критеріїв та альтернатив;

– проставлення оцінок;

– оцінка критеріїв;

– проставлення оцінок порівняння альтернатив по кожному з критеріїв;

– отримання результату та визначення найкращої альтернативи по критеріям.

Дійові особи:

– Екперт.

Діаграма варіантів використання, створена на основі аналізу предметної області за допомогою CASE-засобу Rational Rose для аналізу необхідності та проведення рефакторингу (рис. 4.1):

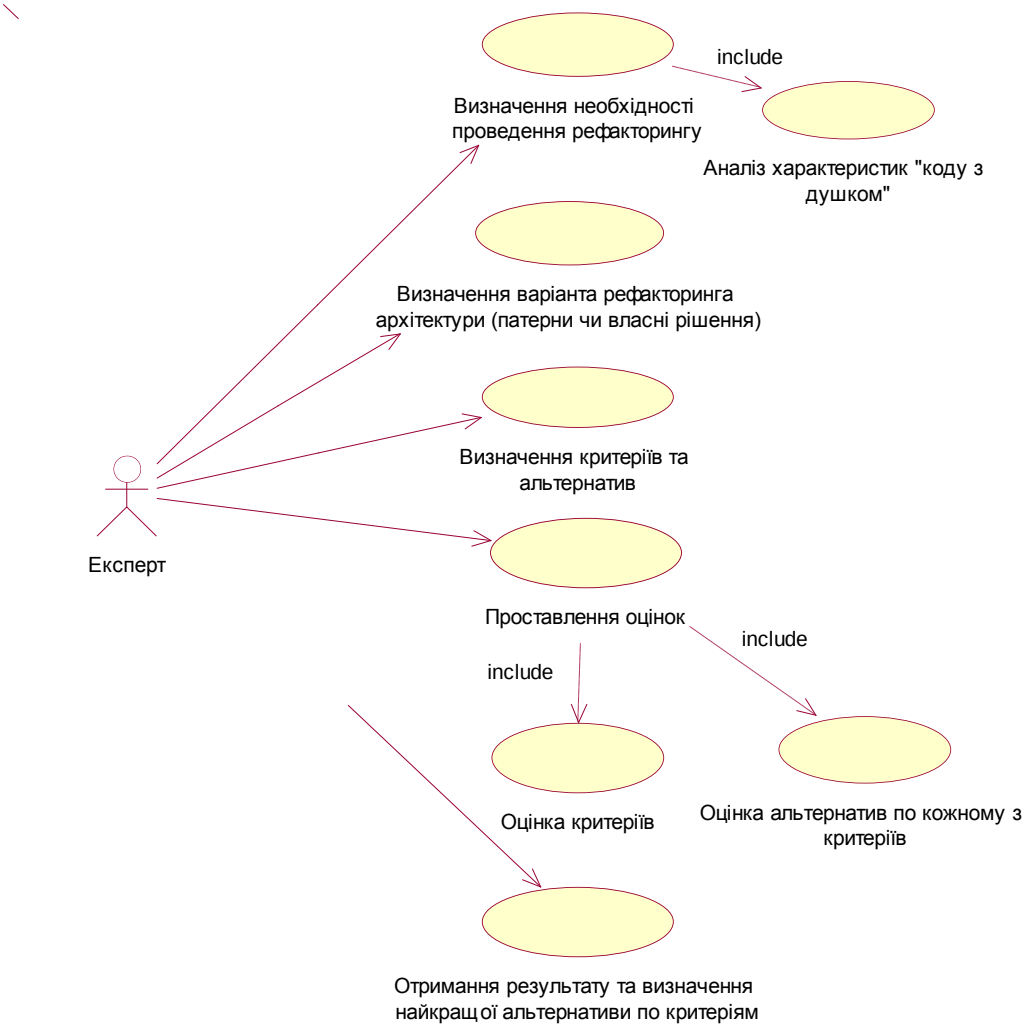

Рис. 4.1. Діаграма варіантів використання

Діаграма класів (class diagram) служить для представлення статичної структури моделі системи в термінології класів ООП. Діаграма класів може відбивати, зокрема, різні взаємозв'язки між окремими сутностями предметної області, такими як об'єкти і підсистеми, а також описує їх внутрішню структуру і типи відносин. На даній діаграмі не вказується інформація про тимчасові аспекти функціонування системи. З цієї точки зору діаграма класів є подальшим розвитком концептуальної моделі проектованої системи.

Діаграма класів є певний граф, вершинами якого є елементи типу "класифікатор", які пов'язані різними типами структурних відносин. Слід зауважити, що діаграма класів може також містити інтерфейси, пакети, відносини і навіть окремі екземпляри, такі як об'єкти і зв'язку. Коли говорять про дану діаграму, мають на увазі статичну структурну модель проектованої системи. Тому діаграму класів прийнято вважати графічним представленням таких структурних взаємозв'язків логічної моделі системи, що не залежать або інваріантні від часу.

Клас (class) в мові UML служить для позначення безлічі об'єктів, які мають однакову структурою, поведінку і відносини з об'єктами з інших класів. Графічно клас зображується у вигляді прямокутника, який додатково може бути розділений горизонтальними лініями на розділи або секції. У цих розділах можуть зазначатися ім'я класу, атрибути (змінні) і операції (методи).

Обов'язковим елементом позначення класу є його ім'я. На початкових етапах розробки діаграми окремі класи можуть позначатися простим прямокутником із зазначенням тільки імені відповідного класу. У міру опрацювання окремих компонентів діаграми опису класів доповнюються атрибутами і операціями.

Передбачається, що остаточний варіант діаграми містить найбільш повний опис класів, які складаються з трьох розділів або секцій. Іноді в позначеннях класів використовується додатковий четвертий розділ, в якому наводиться семантична інформація довідкового характеру або явно вказуються виняткові ситуації.

Діаграма класів, створена допомогою CASE-засобу Rational Rose (рис. 4.2):

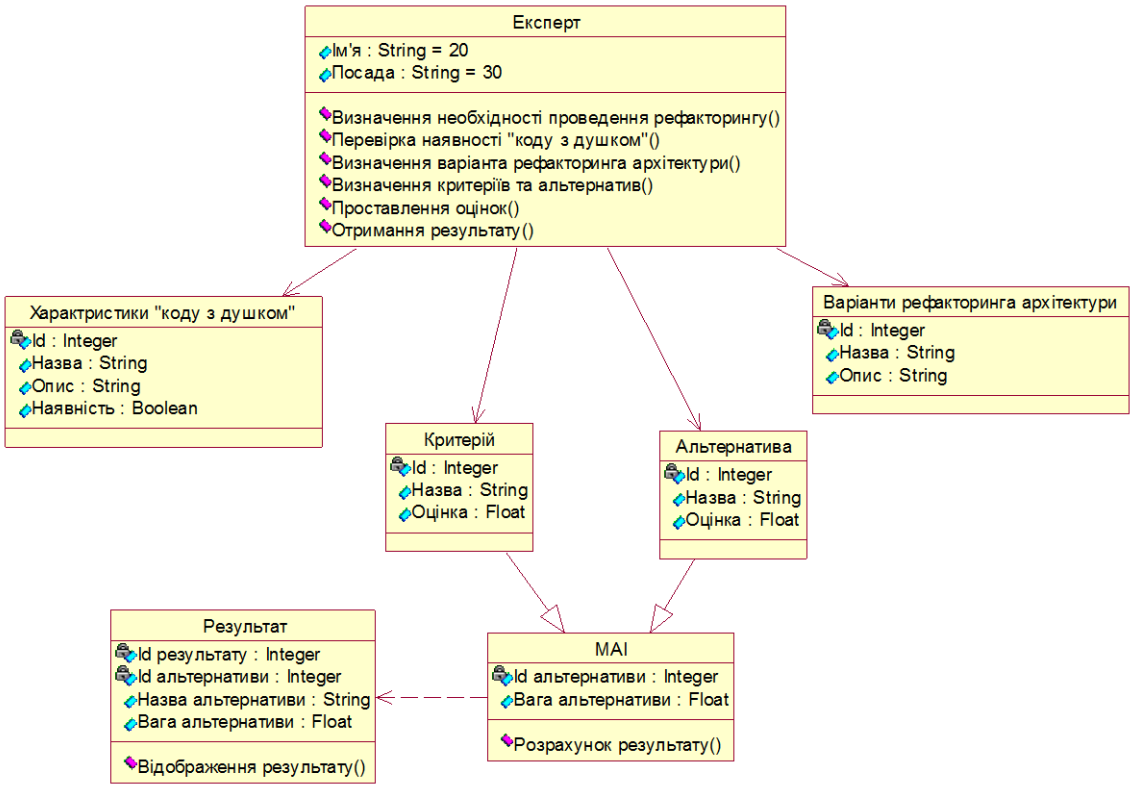

Рис. 4.2. Діаграма класів

Тепер на наступному кроці проектування ми можемо доповнити діаграму варіантів використання діаграмою діяльності.

Діаграма діяльності (або як ще її називають, активності) – відбиває динамічні аспекти поведінки системи. По суті, ця діаграма являє собою блок-схему, яка наочно показує, як потік управління переходить від однієї діяльності до іншої.

Активності на діаграмі "розкидані" по бігових доріжках, кожна з яких відповідає поведінці одного з об'єктів (наприклад, клієнта, менеджера, веб-сервера, сервера БД і т.п.). Завдяки цьому легко визначити, яким з об'єктів виконується кожна з активностей. Доріжка – частина області діаграми діяльності, на якій відображаються тільки ті активності, за які відповідає конкретний об'єкт. Доріжки призначені для розбиття діаграми відповідно до розподілу відповідальності за дії. Ім'я доріжки може означати роль або об'єкт, якому вона відповідає.

Побудуємо діаграму діяльності, використовуючи матеріал 3 розділу признаки необхідності проведення рефакторингу по «коду з душком», для варіанта використання «визначення необхідності проведення рефакторингу» (рис. 4.3):

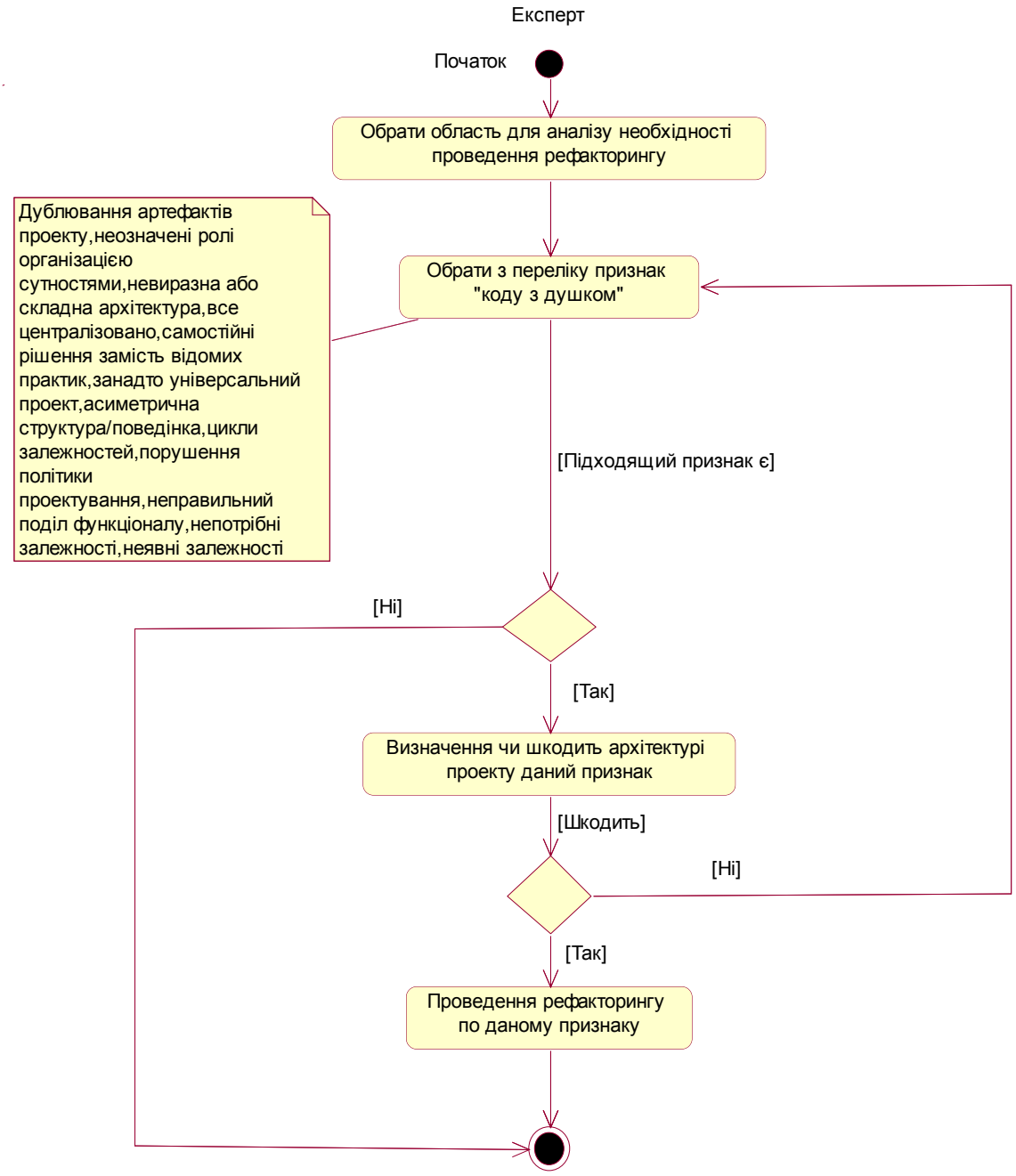

Рис. 4.3. Діаграма діяльності необхідності проведення рефакторингу по наведеним признакам дефекту «коду з душком»

Також побудуємо для варіанта використання «визначення варіанта рефакторинга архітектури (патерни чи власні рішення)» діаграму діяльності (рис. 4.4):

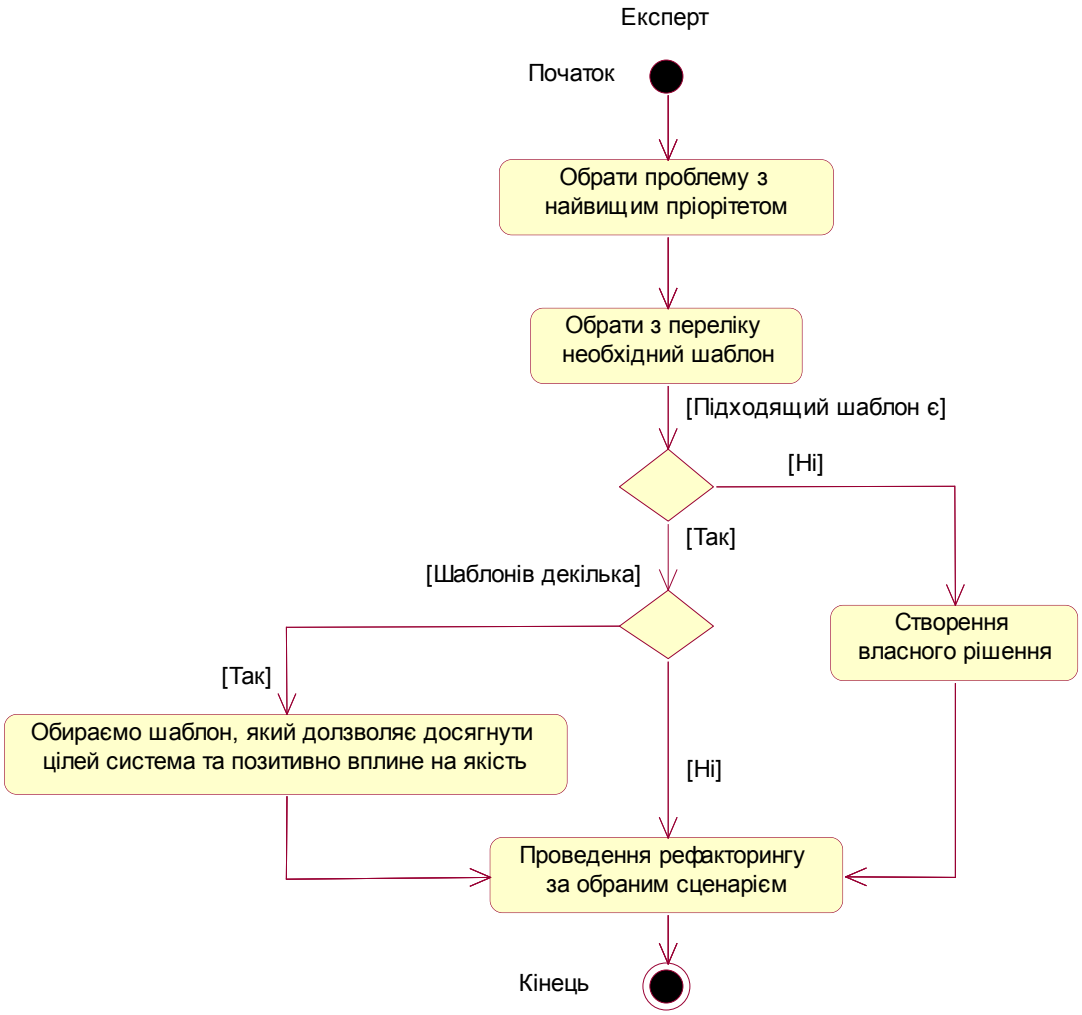

Рис. 4.4. Діаграма діяльності для визначення варіанта рефакторингу архітектури (патерни чи власні рішення)

4.2 Опис методу створення множини альтернативних архітектур

Опишемо процес створення альтернативних архітектур. В процесі проектування необхідно визначити тип архітектури додатку та перелік патернів для заповнення компонентів шарів (модулів). Для кожного компонента треба знайти декілька шаблонів, які виконують ті ж задачі, але мають різні функціональні,

структурні та логічні реалізації. Наведемо приклад патернів шару представлення/компнента UI:

– MVC (Model-view-controller) pattern;
– MVP (Model-View-Presenter) pattern.

За функціональністю вони виконують однаковий функціонал, але мають різну функціональну та структурну реалізацію. MVP є похідним патерном від MVC.

Також як приклад розглянемо патерни шару/компоненти доступу до бази даних (БД):

– DAO (*data access object)* pattern;
– Пряма адресація.

Знову ж за функціональністю ці патерни реалізовують однаково доступ до БД, однак DAO реалізовує кілька шарів абстракції завдяки чому він є більш незалежним від типу БД, гнучким та захищеним, а пряма адресаці до БД є швидшим методом отримання даних, але впротиріч до DAO погано модернізованим та сильно залежним від структури та типу БД.

Припустимо, що ми обрали всі шаблони архітектури, один компонент – один патерн, за винятком компонентів: компоненти UI та доступа до БД, де було обрано по два шаблони, які будуть реалізовувати альтернативи (рис. 4.5.). Тож поєднуючи компоненти ми отримаємо чотири альтернативні архітектури (рис. 4.6).

Після того, як множина альтернативних архітектур створена експертами, необхідно провести попарне оцінювання цієї множини по кожному критерію якості, під час якого отримаємо матрицю попарних порівнянь по критеріям.

Після створення множини альтернативних архітектур експертами проводиться попарне оцінювання сформованої множини по кожному критерію якості. Під час цього оцінювання отримується матриця попарних порівнянь по критеріям. Множина альтернатив виглядать таким чином (рис. 4.5):

1. MVC – Пряма адресація;
2. MVC – DAO;
3. MVP – Пряма адресація;
4. MVP – DAO.

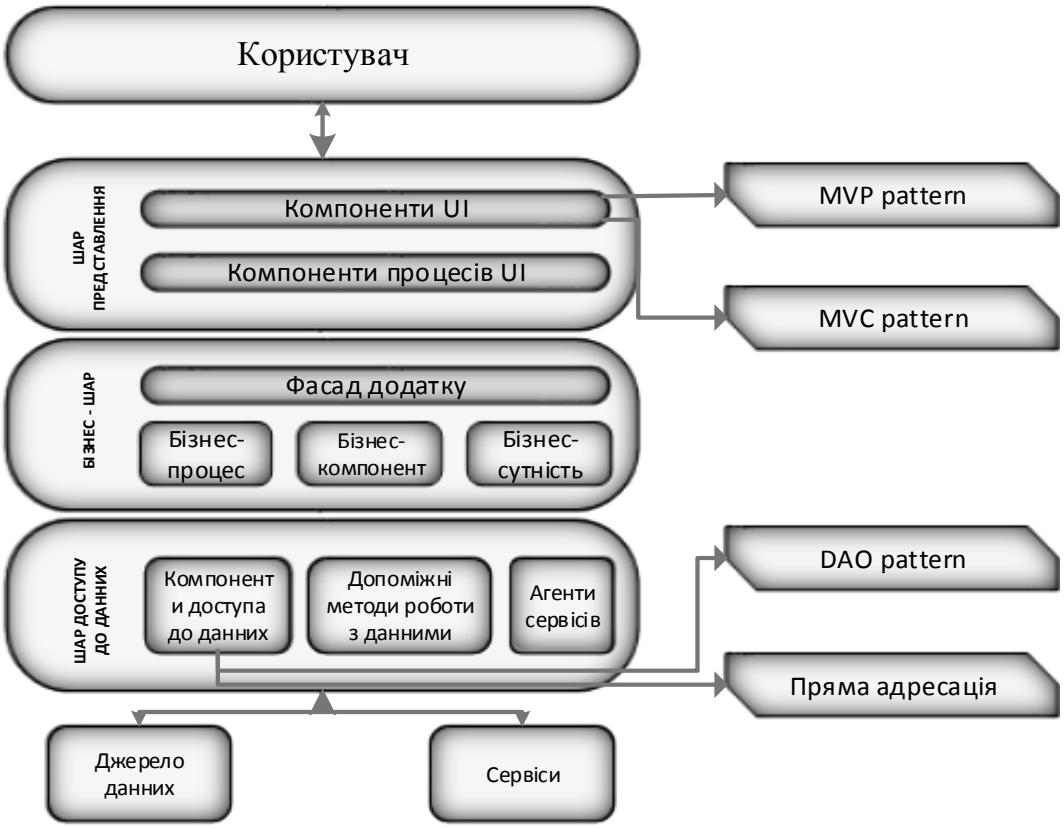

Рис. 4.5. Відношення патернів до компонентів архітектури

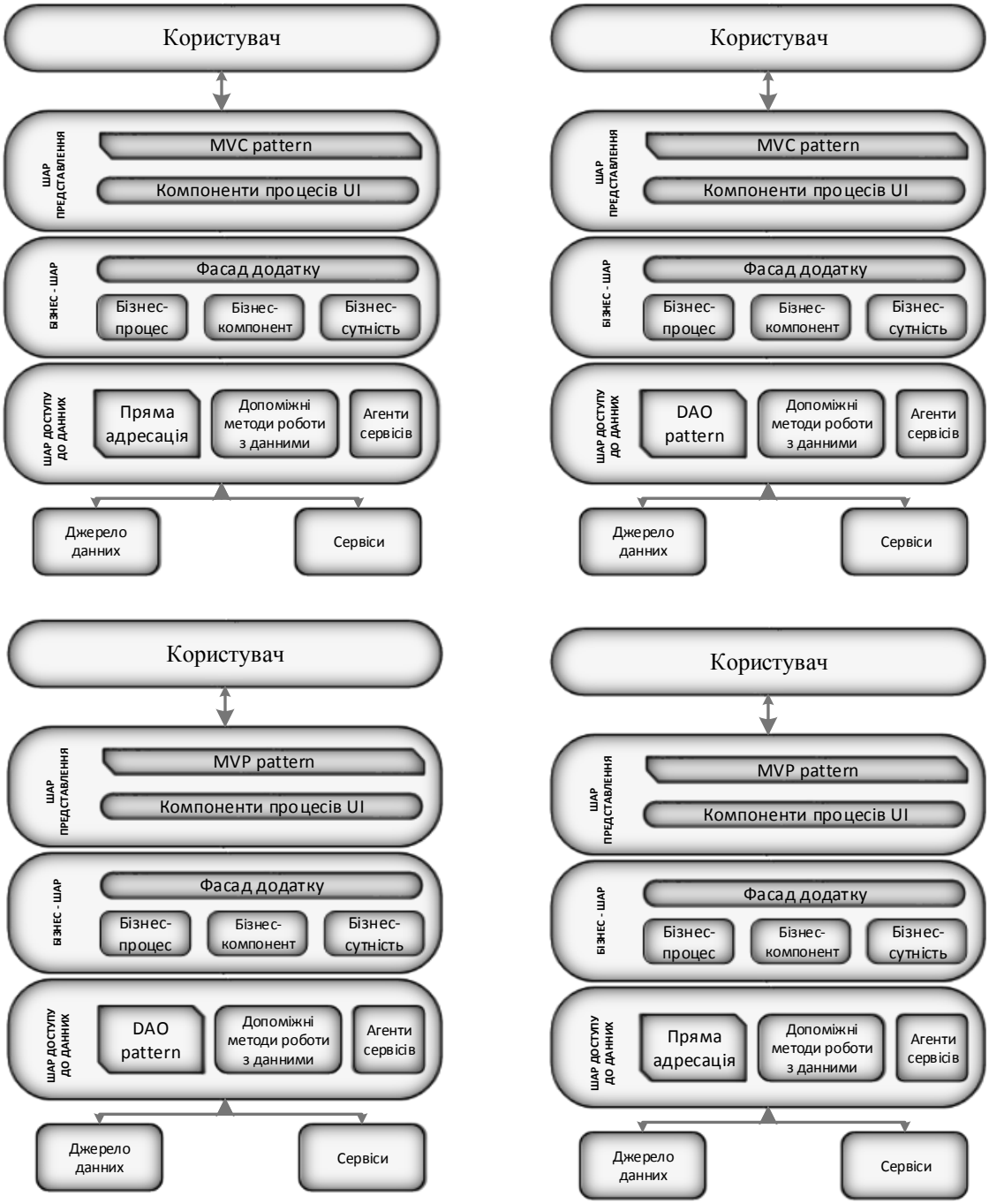

Рис. 4.6. Набір альтернативних архітектур

4.3 Оцінювання альтернативних архітектур методом аналізу ієрархій та прийняття рішень

Процес порівнювального оцінювання архітектур з використанням методу аналізу ієрархій (MAI) представлено на рис. 4.7. Вибір архітектури повинен виконуватись таким чином, щоб побудована на її основі ПС задовольняла вимогам якості. Тому тут представлено два рівні взаємопов’язаних критеріїв якості:

– $K_i^1, i=\overline{1,m1}$ – критерії якості ПС у відповідності зі стандартом ISO/IEC 25010;

– $K_i^2, i=\overline{1,m2}$ – критерії якості архітектури;

– $A_i, i=\overline{1,n}$ – альтернативні архітектурні рішення.

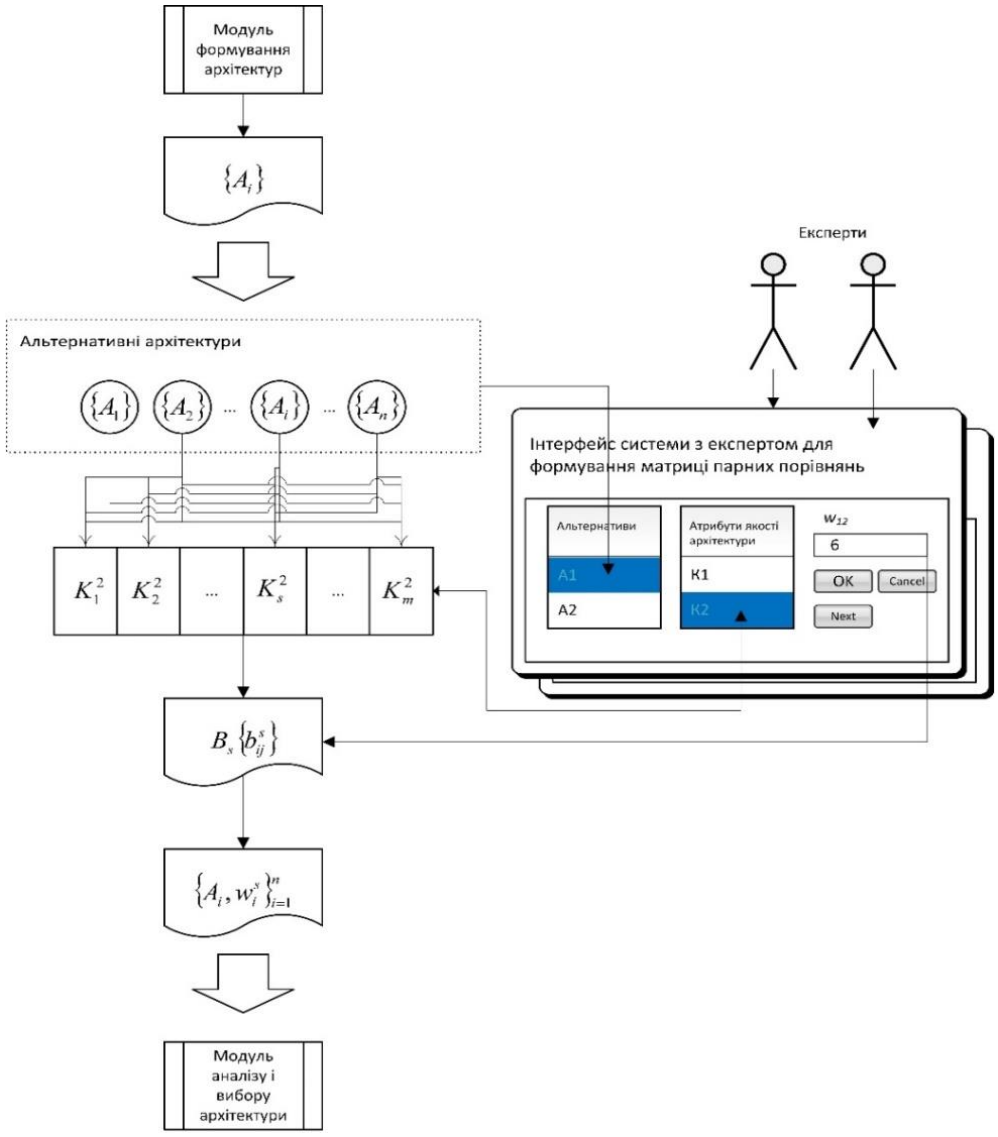

Рис. 4.6. Ієрархічне представлення задачі оцінювання архітектури

Множина критеріїв якості ПС $\{K_i^1\}$, та обмеження на них визначаються при розробці вимог до ПС. А множина критеріїв якості архітектури $\{K_i^2\}$ визначається з використанням методу QFD (Quality Function Deployment, або методу парних порівнянь.

Необхідно вибрати з наявних альтернатив $\{A_i\}$ таку , яка б найкраще забезпечувала якість ПС ,тобто треба вирішити задачу оптимізації за сукупністю критеріїв $\{K_i^1\},\{K_i^2\}$. Це задача багатокритеріальної ієрархічної оптимізації і для її розв'язання будемо використовувати метод аналізу ієрархій Сааті.

При використанні MAI для рішення таких задач відносні оцінки критеріїв (ваги) для альтернатив ω_i^S на кожному рівні знаходяться з використанням матриць парних порівнянь$B_{ij}^S(b_{ij}^S)$ які заповнюють експерти (тут b_{ij}^S визначає перевагу *i*-тої альтернативи над *j*-ю по реалізації s-го критерію).

Коефіцієнти матриць повинні бути узгодженими, тобто $b_{ij} = w_i / w_j \quad \forall b_{ij} \in B$. Ваги в цьому випадку знаходяться як компоненти власного вектору матриці парних порівнянь, які відповідають максимальному характеристичному числу матриці. Обчислення власного вектора матриці є досить трудомісткою процедурою. Тому користуються як правило наближеними співвідношеннями:

$$w_i = \frac{1}{n}\sum_{j=1}^{n}\frac{b_{ij}}{\sum_{j=1}^{n} b_{ij}} \tag{4.1}$$

Для прийняття остаточного рішення по вибору архітектури проводиться аналіз результатів, отриманих вище.

Оскільки в розробці ПС беруть участь декілька груп фахівців, які мають різні пріоритети для кожного з атрибутів якості, то визначаються пріоритети кожної з груп шляхом формування ними матриць парних порівнянь, до яких застосовується MAI і знаходяться пріоритети критеріїв $\{p_i^S\}, i = \overline{1,k_2}, s$ – номер групи експертів. Компромісне рішення приймається як середнє геометричне $p_{ij}^* = \sqrt[n]{p_{ij}^1 \cdot p_{ij}^2 \cdot \ldots \cdot p_{ij}^n}$, або як усереднене з врахуванням показника компетентності груп експертів $p_{ij}^* = p_{ij}^{\alpha_1} \cdot p_{ij}^{\alpha_2} \cdot \ldots \cdot p_{ij}^{\alpha_n}$, ($\alpha_1,\alpha_2,\ldots,\alpha_n$ – показники компетентності).

Якщо потрібно визначити ранжування альтернатив відносно глобальної якості ПС, то необхідно визначити пріоритети критеріїв якості ПС $\{p_i^2\}$, $i = \overline{1,m1}$, застосувавши модифіковану процедуру MAI.

Визначити ваги альтернатив відносно реалізації глобального критерію якості ПС можна за значенням показника:

$$J_i^0 = \sum_{s=1}^{m1} J_i^{1s} \cdot P_s^2, \quad i = \overline{1,n}. \tag{4.2}$$

Тоді показником важливості альтернативи A_i по множині критеріїв буде:

$$J_i = \sum_{j=1}^{m2} P_j \cdot w_j^i, \quad i = \overline{1,n}, \tag{4.3}$$

і ранжування $\{A_i\}$ проводиться за значеннями $\{J_i\}$.

Отримані проранжовані як по окремих показниках якості, так і по їх сукупності, альтернативні архітектури використовуються для прийняття остаточного рішення по вибору варіанта архітектури, яка використовується для рефакторингу з використанням патернів.

Повернемося до прикладу на рис. 4.6. Заповнимо матрицю порівнянь по декільком критеріям, а саме модернізованості (табл. 4.1) та швидкодії (табл. 4.2). Розглянемо модернізованість MVC. Цей патерн є більш широким та менш спеціалізованим, якщо порівнювати з MVP, тож архітектури з MVC краще модернізуються. Якщо порівняти патерни DAO та Пряму адресацію можна прийти висновку, що Пряма адресація майже не піддається модернізації, а DAO – є патерном що направлений на модернізацію.

Заповнимо таблицю попарних порівнянь табл.4.1. таким чином, що якщо проставляється 1 – то такі архітектури є однакові по даному критерію, 9 – сильна перевага першої перед другою.

Таблиця 4.1

Матриця попарних порівнянь по модернізованості

	1	2	3	4
1	1	1/6	1/2	1/5
2	6	1	5	2
3	2	1/5	1	1/6
4	5	1/2	6	1

Розглянемо критерій швидкодії: MVC та MVP мають однакову швидкодію, а DAO – є патерном що реалізовує декілька шарів абстракції, за рахунок чого його швидкодія нижча за Пряму адресацію, яка напряму працює з БД. Заповнимо таблицю попарних порівнянь табл.4.2.

Таблиця 4.2

Матриця попарних порівнянь по швидкодії

	1	2	3	4
1	1	2	1	2
2	1/2	1	1/2	1
3	1	2	1	2
4	1/2	1	1/2	1

Дані з такої матриці (див. табл.4.1. та табл.4.2.) є не аналітичними для сприйняття людиною, тому перетворимо їх у форму аналітичну, що забезпечить зручний аналіз переваг та недоліків окремих архітектурних рішень. Таким чином використовуючи MAI, можна визначити оцінки архітектур, що будуть легко порівнюватися по різним критеріям. Для виконання цієї операції використовується формула:

$$\mathbf{W_i} = \frac{\sum_{j=1}^{n} \mathbf{a_{ij}}}{\sum_{i=1}^{n} \sum_{j=1}^{n} \mathbf{a_{ij}}} \quad (4.4)$$

де W_i – оцінка даної архітектури;

a_{ij} – значення оцінки в матриці попарних порівнянь архітектур (відношення і-ї архітектури до j-й);

n – загальна кількість альтернативних архітектур.

Після виконання усіх математичних операцій з матрицею попарних порівнянь отримується таблиця лінійних оцінок альтернатив табл.4.3.

Таблиця 4.3

Матриця критеріальних оцінок альтернатив

Критерій/архітектура	1	2	3	4
Модернізуються	0,05	0,44	0,098	0,394
Швидкодія	0,33	0,165	0,33	0,165

Таблиця 4.3 виводить матрицю яка показує переваги та недоліки архітектур по певним критеріям, але в такому вигляді незрозуміло, яка архітектура є кращою в поданому ТЗ, тому треба вияснити пріорітети (ваги) конкретних критерієв та зробити приведення критеріальних оцінок до єдиної комплексної оцінки. Це можна провести шляхом заповнення матриці попарних порівнянь для критерієв (табл. 4.4), після чого використовуючи MAI, знайти ваги окремих критеріїв (табл. 4.5). Після цього шляхом лінійної згортки знайти комплексний критерій для архітектур (табл. 4.6).

Таблиця 4.4

Матриця попарних порівнянь критеріїв оцінювання

Критерій/критерій	Модернізуються	Швидкодія
Модернізуються	1	2
Швидкодія	1/2	1

$$\mathbf{Q_i} = \sum_{\mathbf{j=1}}^{\mathbf{n}} \mathbf{q_{ij}} / \sum_{\mathbf{i=1}}^{\mathbf{n}} \sum_{\mathbf{j=1}}^{\mathbf{n}} \mathbf{q_{ij}}, \quad (4.5)$$

де Q_i – вага даного критерію;

q_{ij} – значення оцінки в матриці попарних порівнянь (відношення і-го критерія к j-тому);

n – загальна кількість критеріїв.

Таблиця 4.5.

Матриця ваг критеріїв оцінювання

Критерій/критерій	Модернізуються
Модернізуються	0,66
Швидкодія	0,33

$$\mathbf{K_i} = \sum_{\mathbf{i}}^{\mathbf{n}} \mathbf{Q_i W_i}, \quad (4.6)$$

де K_i – оцінка даної архітектури;

n – загальна кількість критеріїв оцінювання;

W_i – оцінка даної архітектури по і критерію, вираховувалося в (4.4);

Q_i – вагове значення і критерію, вираховувалося в (4.5).

Таблиця 4.6

Матриця комплексних оцінок альтернатив

Архітектура	1	2	3	4
Оцінка	0,15	0,38	0,17	0,29

Остання таблиця 4.6 дає розуміння того, що 2-а архітектура є кращою перед іншими, при цьому 4-а архітектура не набагато гірша за неї, а перша та 3 архітектури сильно їм програють. Саме по результатам лінійної згортки можна прийняти остаточне рішення.

ВИСНОВКИ ДО РОЗДІЛУ 4

У цьому розділі було наведено спроектований програмний комплекс для рефакторингу архітектури за допомогою шаблонів методом аналізу ієрархій та необхідність проведення рефакторингу за допомогою дефектів «коду з душком». Також можливо визначити проводитися рефакторинг буде за допомогою патернів чи власного рішення, яке використається у випадку, коли жоден з патернів не задовільняє умови якості архітектури.

ВИСНОВКИ

У першому розділі було представлено огляд гнучких методологій, де основна увага приділялася не опису кожної існуючої гнучкої методології, а огляду методологій, які могли надати досить повне визначення гнучкості та гнучкого забезпечення якості. Дане визначення представлено з одного боку як теоретичне, а з іншого – як практичне, що може бути корисним фахівцям, які і філософськи відносяться до даної методології і для фахівців, які розвиваються і практикують регулярно гнучку методологію. Також була представлена методологія оцінювання для кращого розуміння гнучких процесів та розкрити основні принципи діяльності різних процесів цієї методології. Метою цієї методології є досягнення балансу між дотримуванням тільки однієї методології і триматися за всі одночасно, тобто поглянути на всі методології із загальної точки зору і використовувати тільки краще і необхідне у кожній з них. Також були розглянуті недоліки гнучкої методології розробки програмного забезпечення та шляхи подолання проблем, які вже використовуються на практиці.

У другому розділі розглянуто твердження, що виникаюча архітектура може замінити явну архітектурну роботу. В якості передумови для аналізу визначено мету, діяльність та завдання архітектурної роботи. Після ретельної перевірки, вирівнювання, проектування для змін, та проектування для зрозумілості (яке складається з виділення сутності системи та деталізованого проектування) були визначені як архітектурні заходи, які необхідно проаналізувати щодо питання про те, чи може виникаюча архітектура замінити явну архітектурну роботу.

Аналіз діяльності показав, що вирівнювання (регулювання) та проектування для змін не охоплюються циклами архітектури, які складаються з кодування та рефакторингу. Нехтування цією діяльністю порушує мету архітектури (тобто загальне задоволення зацікавлених сторін не буде максимальним, а загальні витрати не будуть мінімізовані протягом всього життєвого циклу відповідної системи).

Виділення сутності («форми») системи також не охоплюється виникаючою архітектурою. Можна дозволити сутності системи використовувати виникаючу

архітектуру, але це також може принести в жертву ціль архітектури, витрачаючи додатковий час і зусилля.

Створення зрозумілого, деталізованого проекту дуже добре охоплюється архітектурою. Оскільки це, безумовно, найбільша діяльність з архітектурних робіт, має сенс використовувати для неї виникаючу архітектуру. Такий підхід краще поширює більшість архітектурних робіт і гарантує наявність потенційно дефіцитних навичок для інших архітектурних заходів, які не охоплюються архітектурою.

Спільний підхід до гнучкої архітектурної роботи був отриманий на підставі попередніх висновків. Він використовує архітектуру, що розроблюється, для деталізованого проектування, в той час як інші дії виконуються явно. Цей підхід добре розподіляє архітектурну роботу по всій команді, і потенційно дефіцитні люди, які мають необхідні навички та досвід, можуть зосередитися на заходах, які не охоплюються виникаючими циклами архітектури. Таким чином, agileвартості найкраще сприймаються шляхом максимізації значення, створеного за допомогою наявних навичок та досвіду.

Розвиток і систематичне поширення системи ПЗ є необхідною, але лише однією стороною монети. Іншою стороною є проблеми, з якими стикаються архітектори ПЗ, так як вони відповідальні за уникання складності архітектури існуючого ПЗ. Для цього вони використовують та посилюють оцінювання архітектури та тестування.

Рефакторинг дозволяє покращення структури архітектури, не змінюючи поведінку системи. Використовується він, коли був виявлений «код з душком» при оцінюванні архітектури, мінімум один раз за ітерацію (мається на увазі у гнучкому або ітеративному процесі розробки). Тому проведення рефакторингу вкрай необхідне до та після додавання нового артефакту до проекту ПЗ або коли архітектори визначають важливі проблеми. Таким чином, можна виявити та усунити неправильні проектні рішення ще на початку, гарантуючи покращення якості в архітектурі ПЗ. Рефакторинг є обов'язковим в TDD.

Але несистематичне проведення рефакторингу може призвести до ще більших проблем – наприклад, коли використовують неправильний патерн рефакторинга,

який не розглядає початкові цілі проведення рефакторинга, неохідні якісні ознаки (аспекти реального часу та інше). Тому рефакторинг перш за все вимагає його проведення систематично.

Рефакторинг архітектури ПЗ залишається важливим і цінним інструментом управління складністю програмних систем, так як їх розмір та складність зростають з кожним роком. І якщо використовувати процес рефакторингу систематично, то він може слугувати для архітекторів гарантією збереження ПЗ у хорошому стані.

У четвертому розділі було підсумовано всі набуті знання, застосовані на практиці та наведено спроектований програмний комплекс для рефакторингу архітектури за допомогою шаблонів методом аналізу ієрархій та необхідність проведення рефакторингу за допомогою дефектів «коду з душком». Також можливо визначити проводитися рефакторинг буде за допомогою патернів чи власного рішення, яке використається у випадку, коли жоден з патернів не задовільняє умови якості архітектури.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Abrahamsson P. Agile software development methods: Review and analysis / Abrahamsson P., Salo O., Ronkainen J., Warsta J. – Finland : VVT Publications, 2002. – 107 pp.
2. Agile Alliance. Manifesto for agile software development. [Electronic resource] – 2001. – Access mode: http://www.agilemanifesto.org / (lastaccess:15.12.2020). – Title from the screen.
3. Avison D. Information systems development: Methodologies techniques and tools / Avison D., Fitzgerald G. – Maidenhead: McGraw-Hill, 2003. – 608 pp.
4. Beck K. Extreme programming explained: Embrace change / Beck K. – Massachusetts : Addison-Wesley, 1999. – 189 pp. Bibliogr.: pp. 10-70.
5. Beck K. Extreme programming explained: Embrace change / Beck K., Andres C. – Boston : Addison-Wesley Professional, 2004. – 216 pp.
6. Boehm B. Balancing agility and discipline: A guide for the perplexed / Boehm B., Turner R. – Boston : Addison-Wesley, 2004. – 300 pp. Bibliogr.: pp. 165-194.
7. Cockburn A. Selecting a project's methodology / Cockburn A. – IEEE Software, 2000. – 71 pp. Bibliogr.: pp. 64-71.
8. Fowler M. The agile manifesto: Where it came from and where it may go. [Electronic resource] – 2006. – Access mode: http://martinfowler.com/articles/agileStory.html / (lastaccess: 07.12.2020). – Title from the screen.
9. Highsmith J. The great methodologies debate: Part 1: Today, a new debate rages: Agile software development vs. rigours software development / J. Highsmith // Cutter IT Journal. – 2001. – №14. – 2-4 pp.
10. Highsmith J. Agile software development: Why it is hot! / Highsmith J. – Cutter Consortium white paper, Information Architects. – 2002. – 236 pp. Bibliogr.: pp. 1-22.
11. Highsmith J. Agile software development ecosystems / Highsmith J. – Boston : Addison-Wesley, 2002. – 404 pp. Bibliogr.: pp. 1-50.
12. Highsmith J. Agile project management / Highsmith J. – Boston: Addison-Wesley, 2004. – 277 pp.
13. Juran J. M. Juran's quality control handbook / Juran J. M. & Gryna F. M. – Maidenhead: Mcgraw-Hill, 1988. – 400 pp.

14. Lindvall M. Empirical findings in agile methods. Proceedings of Extreme Programming and agile Methods / Lindvall M., Basili V. R., Boehm B., Costa P., Dangle K., Shull F., Tesoriero R. Williams, L. & Zelkowitz M. V. – Chicago : XP/agile Universe , 2002. – . Bibliogr.: pp. 197-207.

15. Meyer B. Object-oriented software construction / Meyer B. – New Jersey: Prentice Hall PTR, 2000. – 207pp. Bibliogr.: pp. 4-20.

16. Poppendeick M. Lean software development: An agile toolkit for software development managers / Poppendeick M. & Poppendeick T. – Boston: Addison Wesley, 2003. – 230pp. Bibliogr.: pp. 11-28.

17. Pressman R. S. Software engineering a practitioner's approach / Pressman R. S. – Maidenhead: McGraw-Hill, 2001. – 940 pp.

18. Schwaber K. Agile software development with SCRUM / Schwaber K. & Beedle M. – New Jersey: Prentice-Hall, 2002. – 176 pp. Bibliogr.: pp. 23-30.

19. Sommerville I. Software engineering / Sommerville I. – Boston: Addison-Wesley, 2004. – 816 pp.

20. Turk D., France R., & Rumpe B. Limitations of agile software processes. Proceedings of the Third International Conference on eXtreme Programming and Agile Processes in Software Engineering / Turk D., France R., & Rumpe B, 2002. – 840 pp. Bibliogr.: pp. 43-46.

21. Weisert C. The #1 serious flaw in extreme programming (XP). [Electronic resource] – 2002. – Access mode: http://www.idinews.com/Xtreme1.html 2002. – Title from the sreen.